小眼睛科技

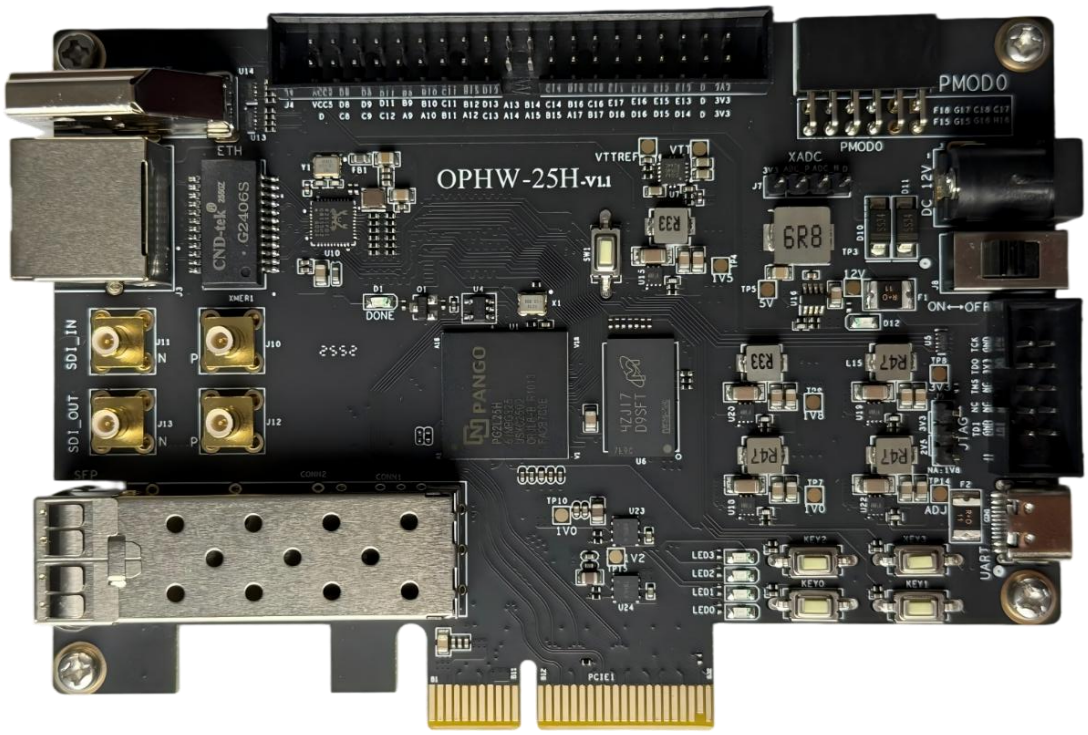
提供一站式 FPGA&嵌入式解决方案

OPHW-25H 开发板实验教程

深圳市小眼睛科技有限公司

文档版本修订记录

版本号	发布日期	修订记录
V1.0	2026/1/7	初始版本



公司名称：深圳市小眼睛科技有限公司

地址：深圳市宝安区西乡街道F518时尚创意园

官方网址：www.meyesemi.com

官方淘宝店铺：小眼睛半导体

B站：小眼睛半导体（视频教程免费学）

* 加入FPGA开发者技术交流与5000+FPGA开发者实时沟通

QQ2群： 442106123 QQ3群：882634519）

*配套资料下载、技术答疑请登录逻辑矩阵技术论坛

LOGIC MATRIX 逻辑矩阵

逻辑矩阵技术论坛欢迎各位发烧友加入
让我们共建开源生态，持续赋能行业发展
https://www.szlogicmatrix.com/

*扫码注册开源技术论坛

*扫一扫关注官微

* 官方旗舰店

目录

1.FPGA 开发工具使用

1. 1. 实验简介

实验目的：

PDS软件的安装和环境搭建。

实验环境：

Window11

PDS2022.2

硬件环境：

OPHW-25开发板

1. 2. 实验原理

1.2.1.PDS 软件的安装

1.2.1.1.软件简介

Pango Design Suite 是紫光同创基于多年 FPGA 开发软件技术攻关与工程实践经验而研发的一款拥有国产自主知识产权的大规模 FPGA 开发软件，可以支持千万门级 FPGA 器件的设计开发该软件支持工业界标准的开发流程，可实现从 RTL 综合到配置数据流生成下载的全套操作。

1.2.1.2.支持平台

软件工具	Windows 操作系统
ADS 综合工具、OEM 综合工具、PDS 后端工具	Windows 7,10,11、

1.2.1.3.软件安装

所有版本的安装均一致，文中以PDS_2022.1版本为例，将软件安装在C:\pango\PDS_2022.1（软件默认安装路径），若选择自定义安装路径需注意路径不可出现中文和特殊字符。软件安装完成后，会在桌面以及程序菜单中添加快捷方式 PangoDesign

Suite2022.1；在程序菜单 Pango Design Suite2022.1 文件夹中包含 Pango Design Suite、软件卸载的快捷方式 Uninstall、程序附件 Accessories 以及软件文档 Documents。

本章节安装流程以 Pango Design Suite 2022.1 Windows 版本安装进行说明，下面将详细介绍PDS安装过程。

1.2.1.4.安装程序

首先关闭电脑所有杀毒软件，否则杀毒软件有可能会拦截一些组件，造成安装失败或功能缺失等不确定结果；将压缩包解压出来（注意：解压目录的路径名称只能够包含字母、数字、下划线，不要出现中文与特 殊字符，否则安装程序有可能出问题）

双击安装包中的安装程序 Setup.exe，启动安装程序如下图所示：

图 1.2-1

启动安装程序后的界面如下：

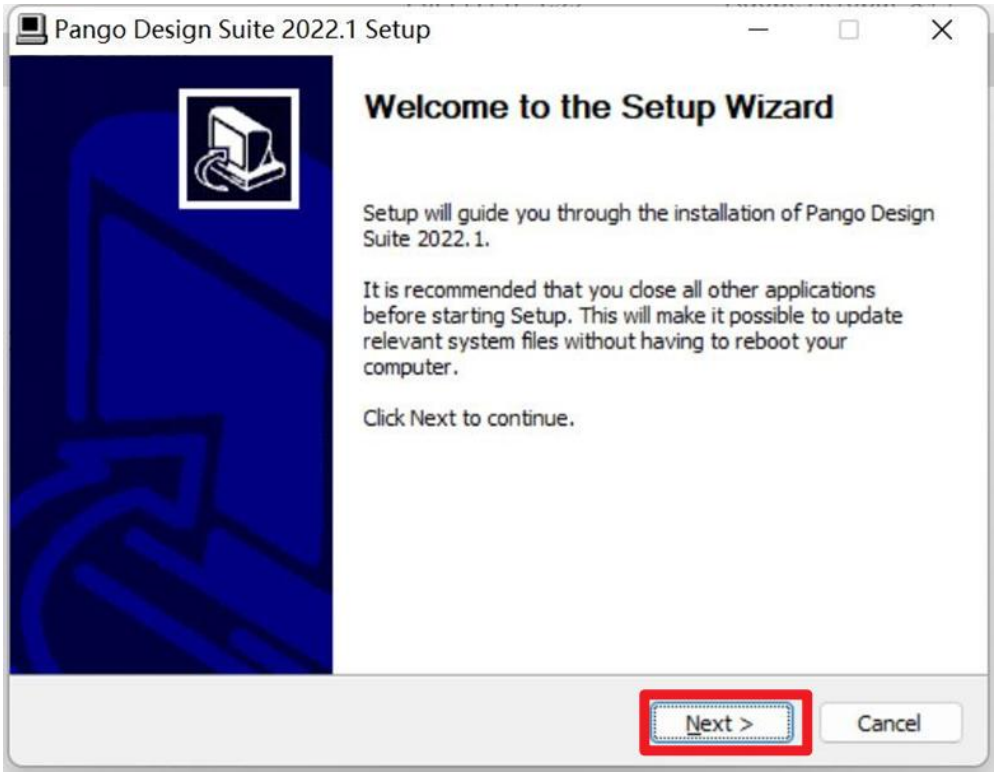


图 1.2-2

点击“Next”，跳转至许可协议对话框：

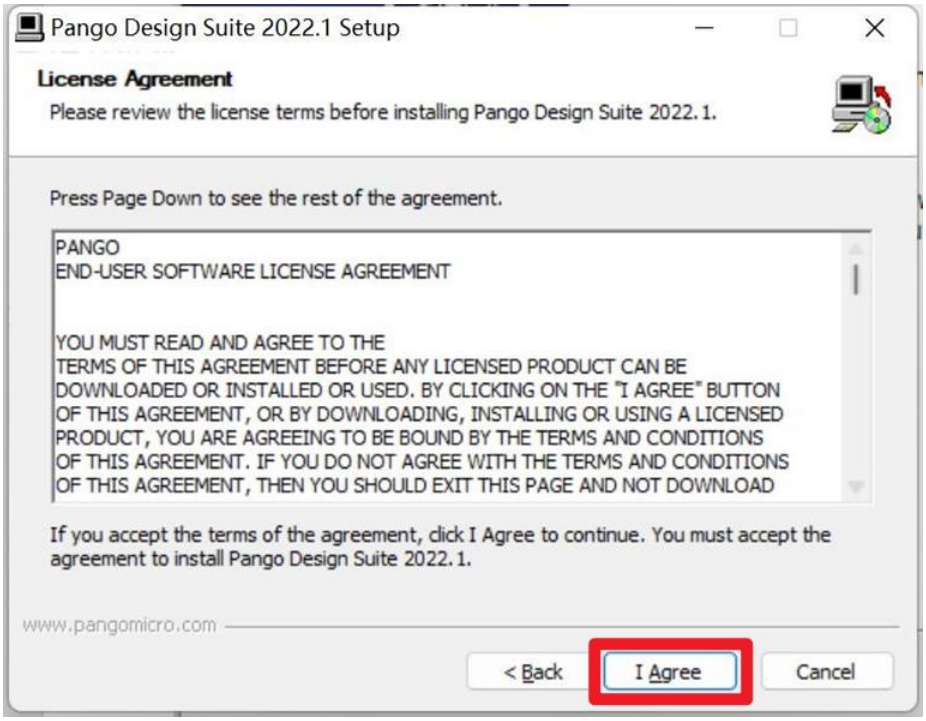


图 1.2-3

选择接受许可协议，点击“I Agree”按钮，进入选择安装路径选择框，如下图所示，默认安装路径为C:\pango\PDS_2022.2，建议采用默认路径，若使用自定义安装路径需注意路径不要出现中文和特殊字符。

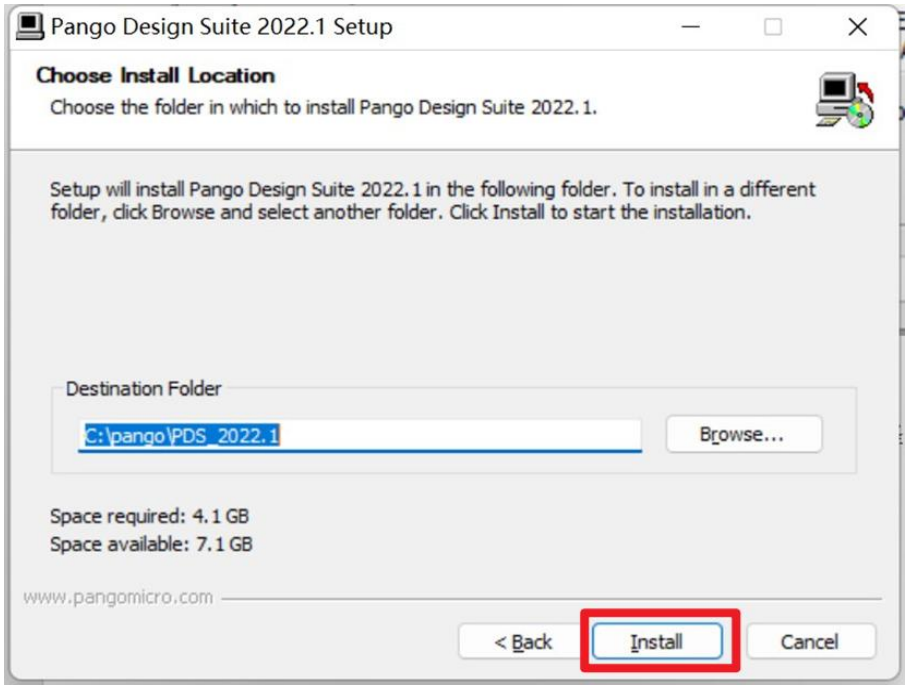


图 1.2-4

直接点击“Install”，则跳转到安装界面。

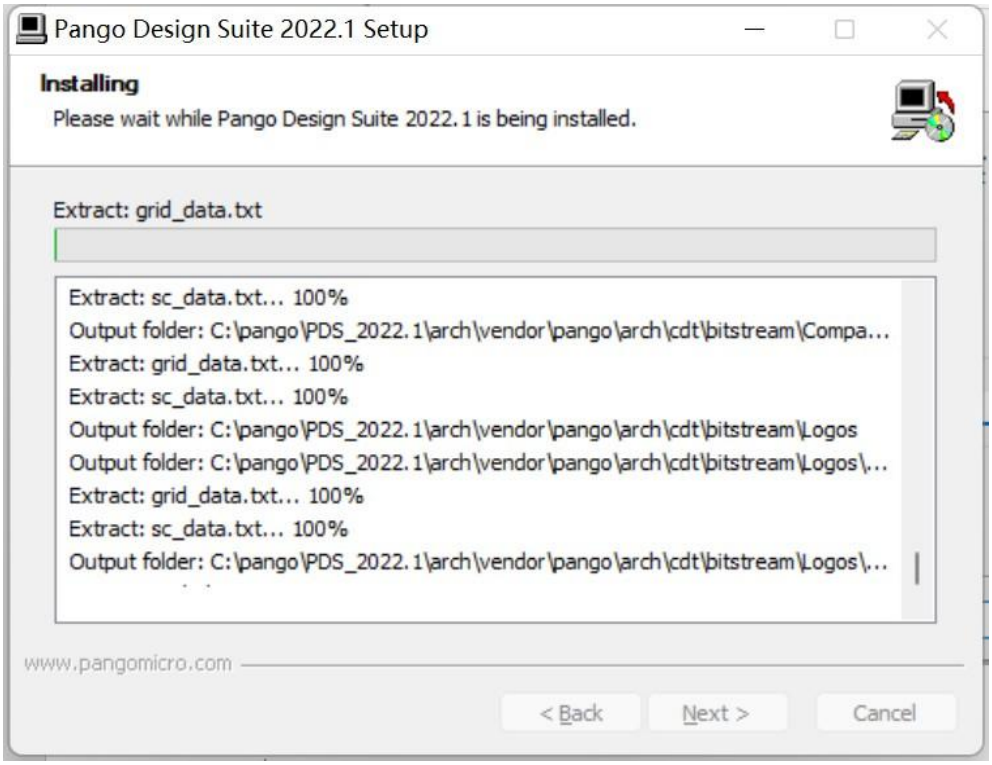


图 1.2-5

耐心等待至完成全部安装过程，点击“Finish”，安装完毕。

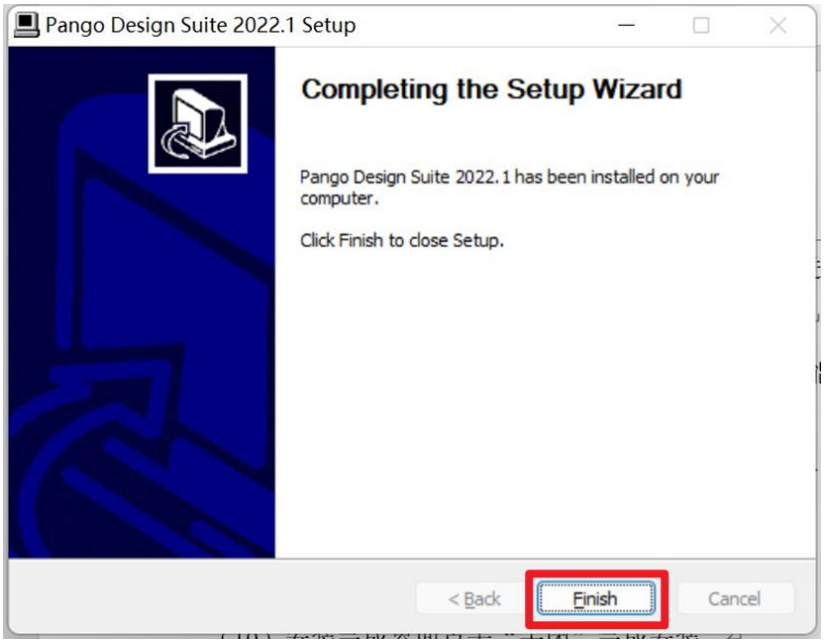


图 1.2-6

安装完成后，会提示是否需要安装运行库 vcredist_VS2017.exe。若电脑之前未安装过则需要安装此运行库后才能运行 PDS，点击“是”按钮进行安装；若电脑之前已安装过此运行库则无需再次安装，点击“否”按钮不进行安装即可。（如果没有这个提示，可以不管）

注：如果不确定，建议点击“是”进行安装，否则可能导致 PDS 无法运行。

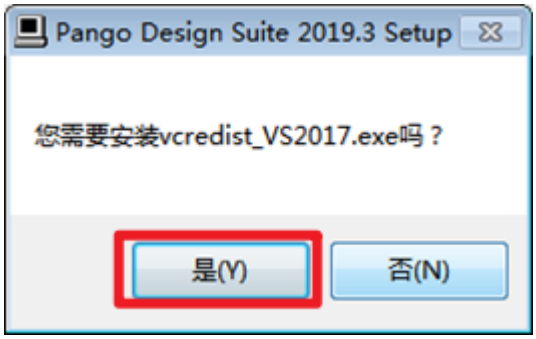


图 1.2-7

点击“是”进入运行库安装界面，选择同意许可条款和条件，点击安装按钮进行安装。

图 1.2-8

安装完成界面点击“关闭”完成安装。

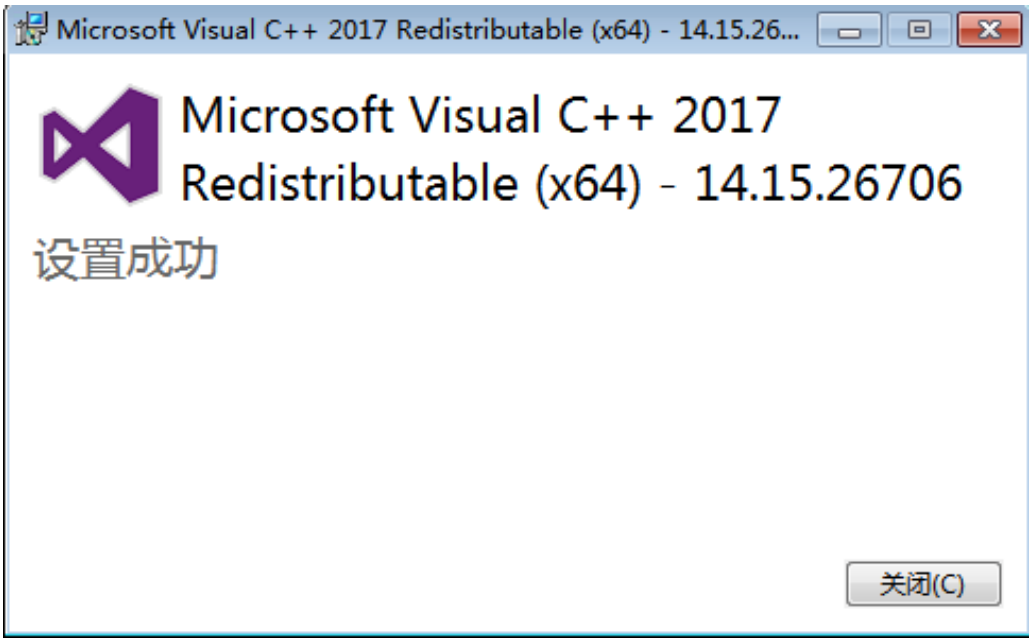


图 1.2-9

完成安装后，会提示是否需要安装 USB Cable Driver。安装点击“是”，否则可能导致下载器无法正常使用。

图 1.2-10

点击“是”进入驱动程序安装界面：

图 1.2-11

点击“下一步”进入安装驱动程序许可协议界面。点击“我接受这个协议”，然后点击”下一步“。

图 1.2-12

点击“完成”，完成驱动程序的安装。

图 1.2-13

完成安装后，会提示是否需要安装 ParellelPortDriver。安装点击“是”。ParellelPortDriver 的安装程序保存地址：PDS 软件安装目录\parellel_driver\Win32\InstallDriver.exe

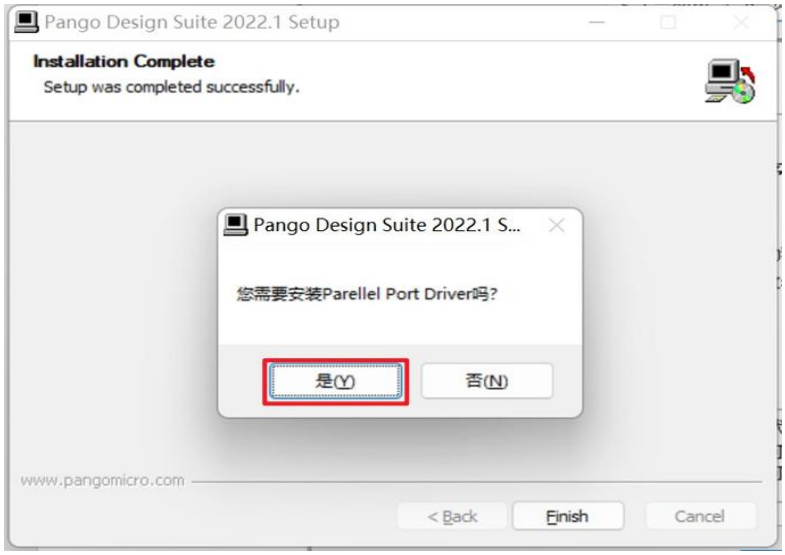


图 1.2-14

点击“是”进行并口驱动程序安装。

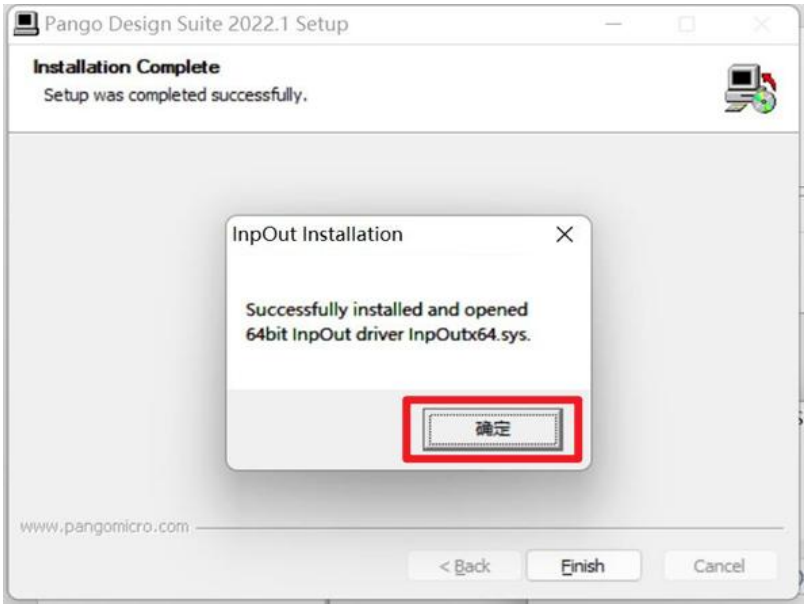


图 1.2-15

完成后点击“确定”，结束安装。在桌面上看到如下图标：

图 1.2-16

1.2.1.5.License 关联，环境变量设置

软件完成安装后，Pango Design Suite 需要 License 文件才能正常使用，若安装Lite版本软件无需License。License文件可联系相关供应商或客服获取。

本教程使用软件版本开发语言支持 Verilog，若使用 VHDL需更新支持VHDL的软件版本

为方便管理 license 文件，建议在 PDS 软件安装目录下新建一个 license 文件夹存放 license 文件。

首先在电脑上打开运行窗口(win+r)，接着在窗口内输入 sysdm.cpl 然后回车。在系统属性界面内选择高级，然后点击环境变量，进行设置。

图 1.2-17

或者直接打开系统环境变量

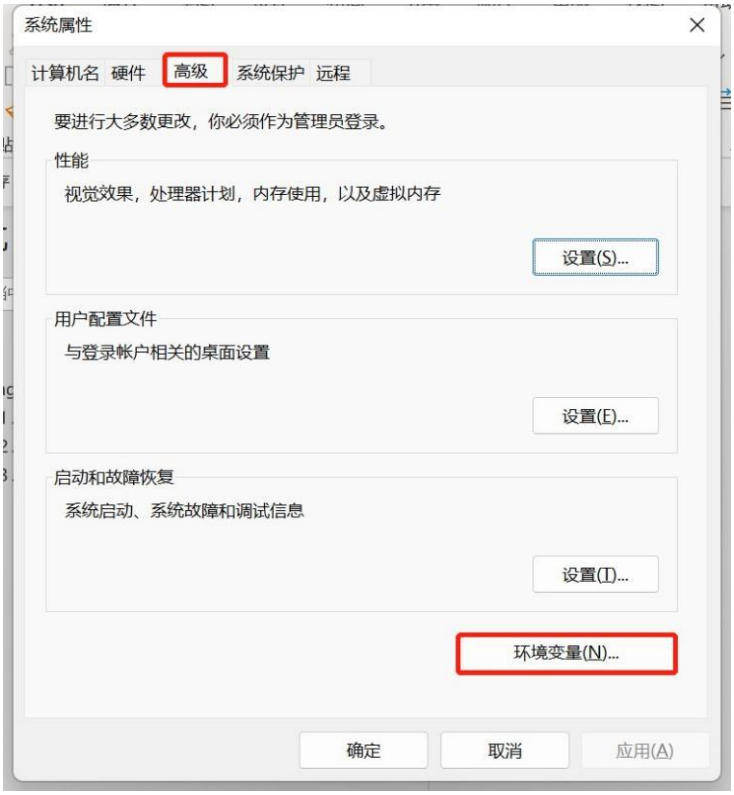


图 1.2-18

1.2.1.6.PDS license 环境变量设置

查看下载的资料包内PDS license 文件路径，建议将license文件放置在PDS软件文件夹地址内，这里以：D:\pango\license\pds_node-locked.lic地址为例设置环境变量：

变量名：PANGO_LICENSE_FILE

变量值：D:\pango\license\pds_node-locked.lic

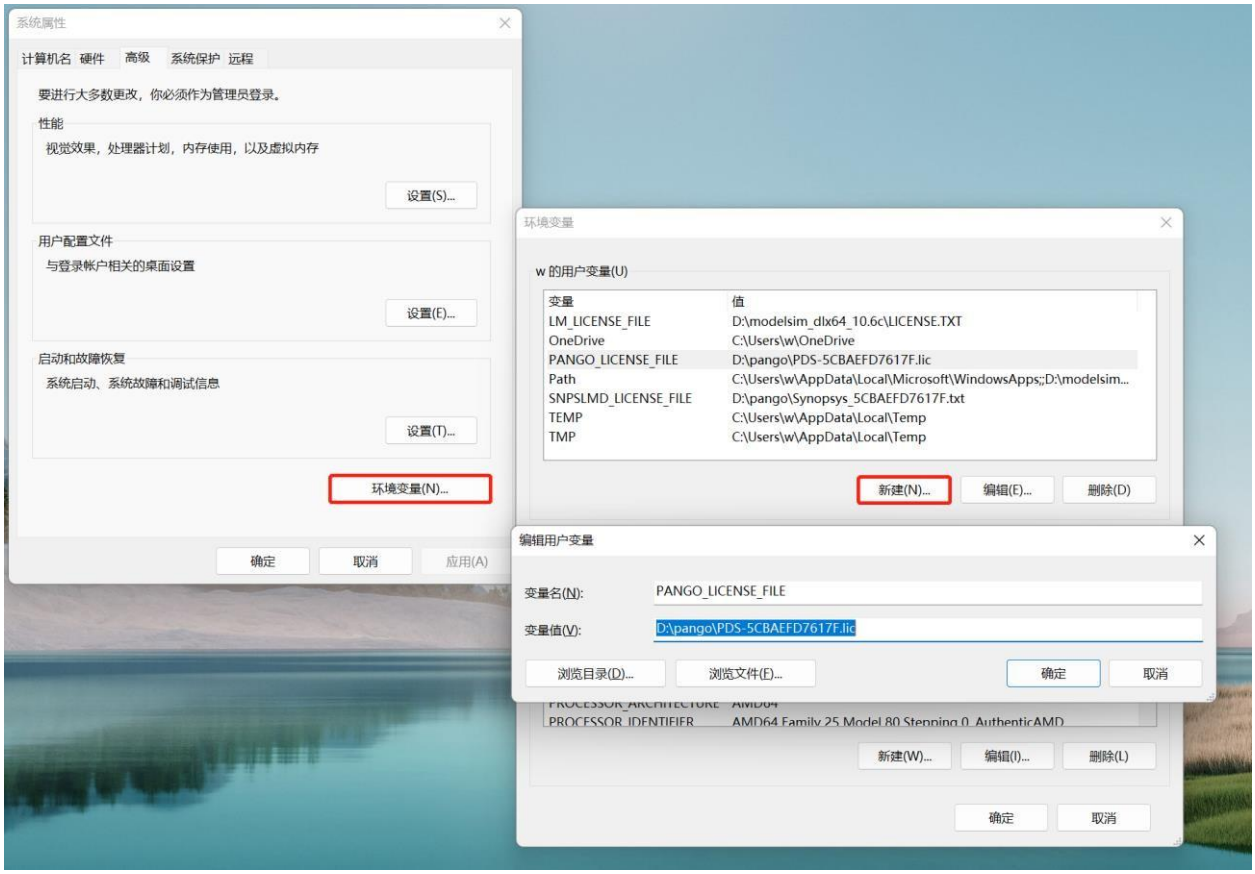


图 1.2-19

如果使用开发板资料提供的通用License，还需要安装tap-windows软件。其主要作用是用来生成虚拟网卡，具体如下所示：

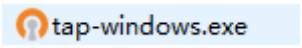


图 1.2-20

双击运行，全部保持默认。

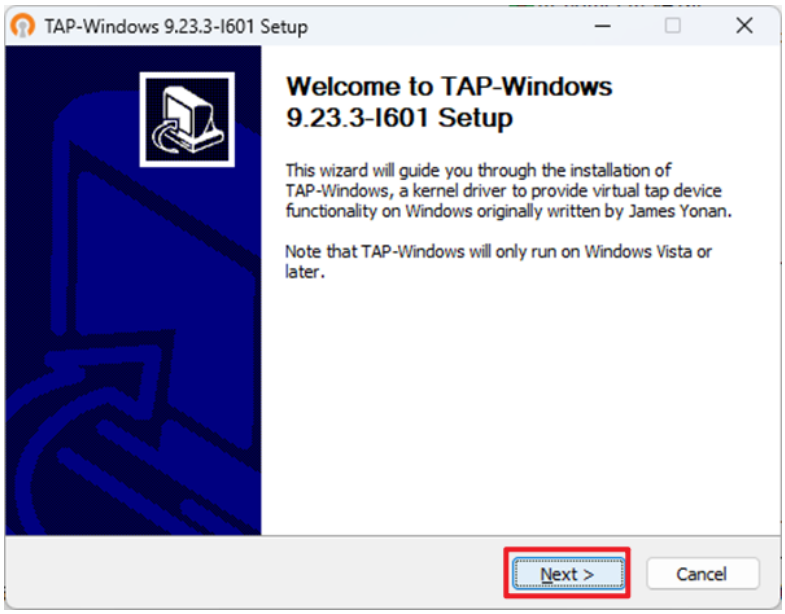


图 1.2-21

点击Next。

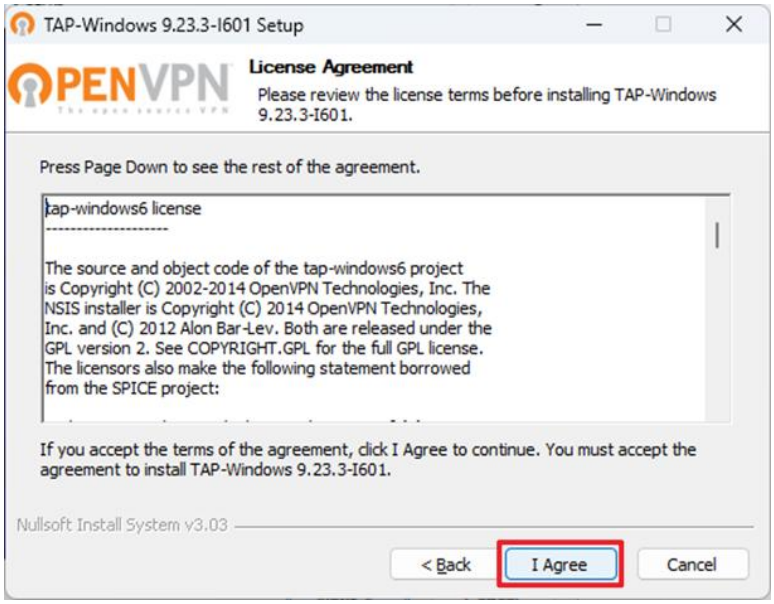


图 1.2-22

点击I Agree。

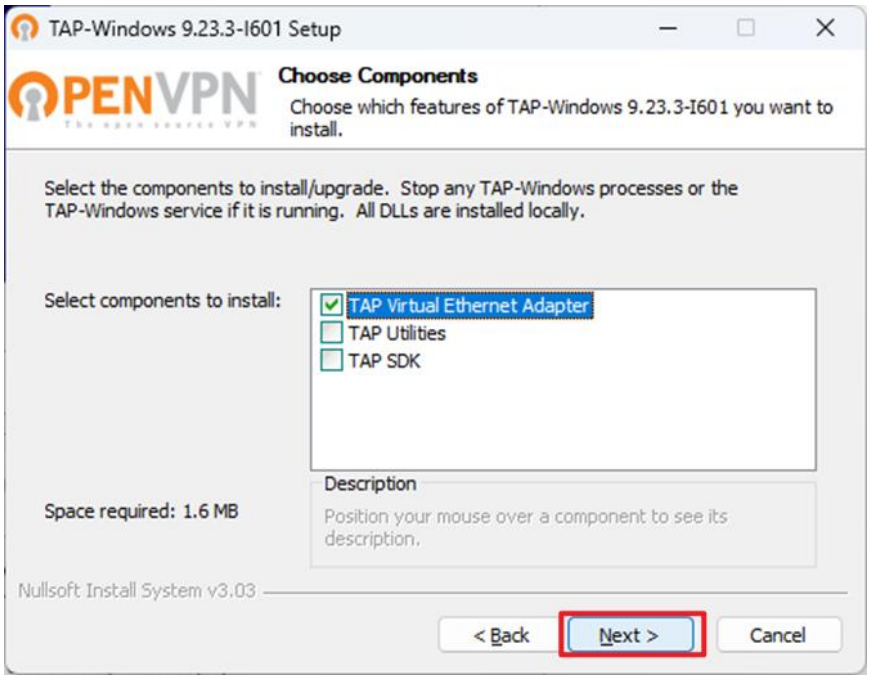


图 1.2-23

保持默认，点击Next。

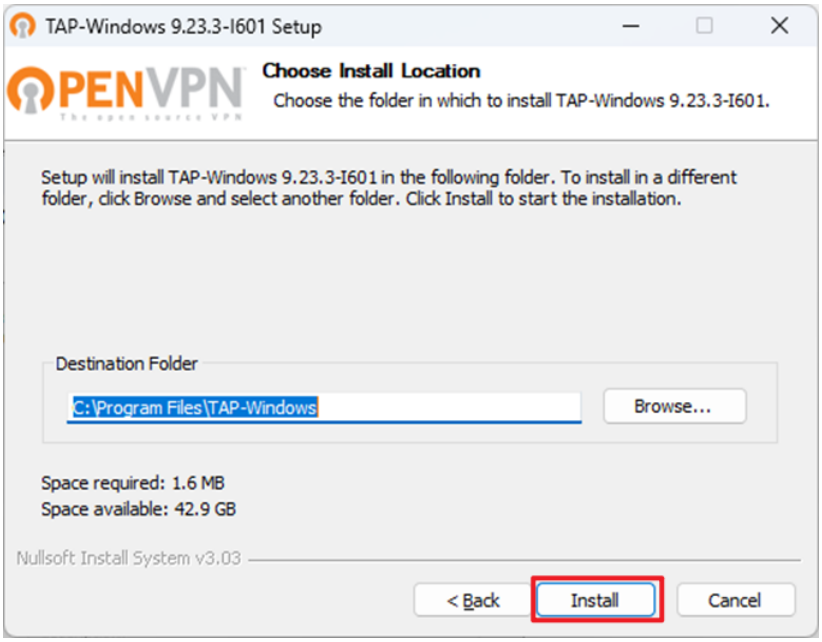


图 1.2-24

安装路径保持默认，点击Install，等待安装完成。

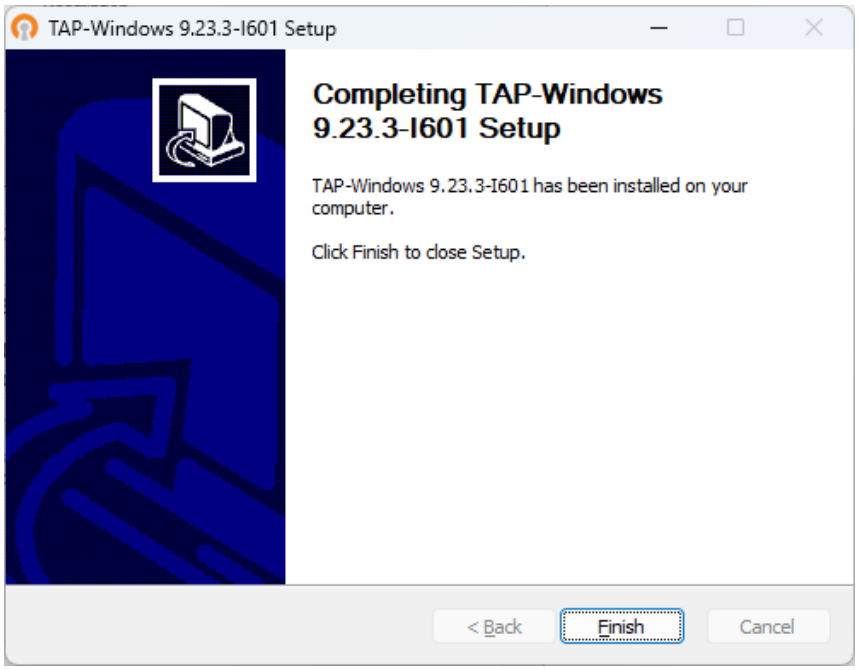


图 1.2-25

点击Finish，至此，安装完成。

打开设备管理器，在网络适配中找到TAP-Windows Adapter V9，并双击。

图 1.2-26

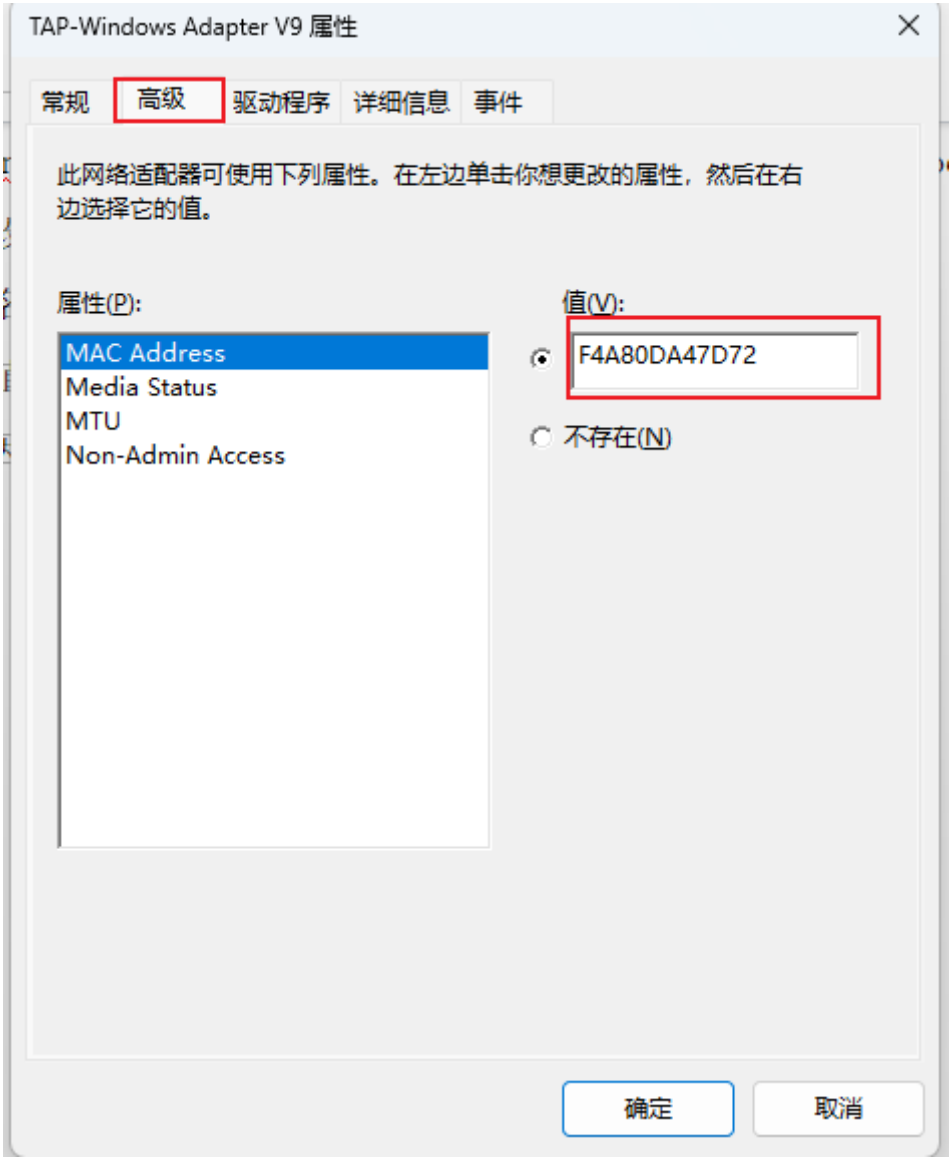
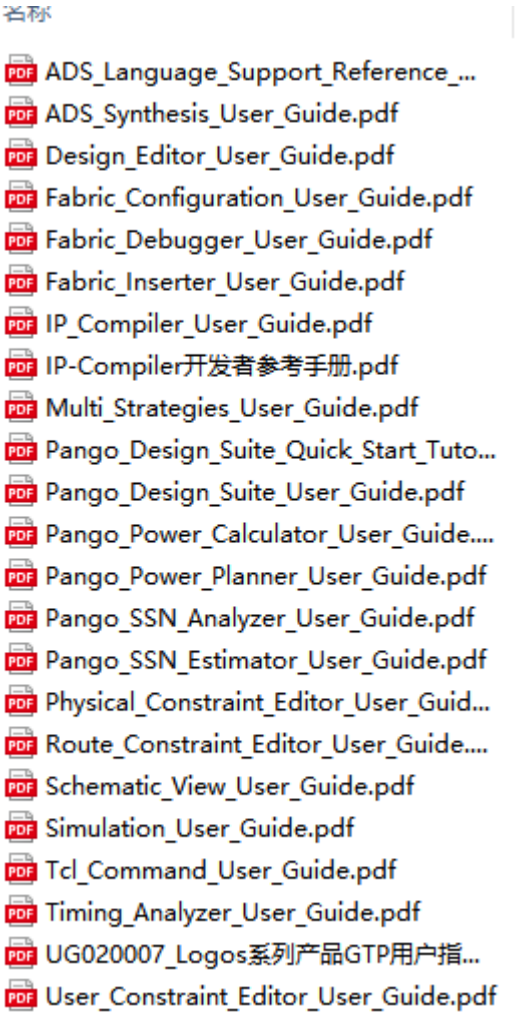


图 1.2-27

选择属性选项卡，并将MAC Address的值改成Lincese文件名后面的字符串。

1.2.2.PDS 工具的使用

详见开发板配套资料《PDS快速使用手册》文档,或软件安装目录doc文件夹下《Pango_Design_Suite_Quick_Start_Tutorial.pdf》和《Pango_Design_Suite_User_Guide.pdf》。



2.Modelsim 的使用和 do 文件编写

2.1. 实验简介

实验目的：

了解Modelsim的基本使用方法，完成do文件的编写，提高仿真效率。

实验环境：

Window11

PDS2022.2

Modelsim10.6c

2.2. 实验原理

将Modelsim的命令编写到一个do文件中，这样每次仿真时，只需运行这个do文件脚本即可自动执行其中的所有命令，从而显著提高重复仿真的效率。

2.3. 接口列表

暂无

2.4. Testbench 文件的编写

Testbench文件其实就是模拟信号的生成，给我们所设计的模块提供输入，以便测试。因为我们上板去生成比特流，尤其是比较复杂的算法，往往是需要耗很多时间的。所以要快速验证我们的设计逻辑是否正常，还得是用仿真来验证，不管是模拟图像的生成还是信号的生成，都可以通过Testbench来完成，但是，要注意一点，逻辑前仿真通过了只能说明80%上板没问题，剩下的可能就要看实际的时序了，毕竟仿真是理想状态，实际总是不太理想。

接下来介绍Testbench的基本编写方法：

`timescale 1ns/1ns该语句第一个1ns表示时间单位为1ns，第二个1ns表示时间精度为1ns。注意的是，时间单位不能比时间精度还小。时间单位表示运行一次仿真所用的时间。时间精度表示仿真显示的最小刻度。

#10表示延时10个单位时间，比如`timescale 1ns/1ns，#10表示延时10ns。

initial对信号进行初始化，只会执行一次。

{$random}%x，表示随机取[0,x-1]之间的数字。x为正整数。如果是$random%x，则是[-(x-1),x-1]的数。

$display 打印信息，会自动换行。

$stop 暂停仿真。

$readmemb读取文件函数。

$monitor为监测任务，用于变量的持续监测。只要变量发生了变化，$monitor就会打印显示出对应的信息。

输入信号一般用reg定义，方便后续用always块生成想要模拟的值，输出一般直接wire引出即可。

例如生成时钟，always#10 sys_clk = ~sysclk; 表示每10个单位时间就翻转一次，如果时间单位是ns，那就是每10ns翻转一次，就是生成了50MHZ的时钟。周期是20ns。

接下来给出一个参考的testbench，如下所示：

```
1.  `define UD #1
2.  module tb_led_test();
3.
4.  reg              clk        ;
5.  reg              rst_n      ;
6.  wire[7:0]     led        ;
7.  reg [7:0]     data       ;
8.
9.  initial begin
10.      rst_n <= 0;
11.      clk     <= 0;
12.      #20;
13.      rst_n <= 1;
14.      #2000
15.      $display("I am stop"); //
16.      $stop;
17. end
18. always#10 clk = ~clk;//20ns    50MHZ
19.
20. led_test
21. #(
22.      .CNT_MAX       (10)
23. )u_led_test(
24.  .clk            (clk       ),// input
25.  .rstn           (rst_n     ),// input
26.  .led            (led       ) // output [7:0]
27. );
28.
29. initial begin
30.      $monitor("led:%b", led);
31. end
32.
33. always@(posedge clk or negedge rst_n)       begin
34.      if(!rst_n)
35.           data       <=       8'd0;
36.      else
37.      begin
38.           data       <=       {$random}%256;
39.           $display("Now data is %d",data);
40.      end
41. end
42.
```

```
43.  endmodule
```

本testbench模块tb_led_test用于对LED控制模块led_test进行仿真验证。其主要功能包括：产生50MHz的时钟信号，提供上电复位信号（20ns后释放），实例化待测模块并传递参数CNT_MAX=10，同时在每个时钟周期生成一个随机数data并打印输出，以辅助观察模块运行情况。此外，testbench会实时监控LED输出信号的变化，并在2000ns时$display打印提示信息后自动结束仿真。注意$stop仅仅是暂停仿真，不是完全结束仿真，还可以通过run指令继续运行仿真。

在信号设置方面，clk为1位寄存器类型信号，用于产生50MHz的时钟；rst_n为1位寄存器类型信号，用于提供低电平有效的复位控制；led为8位线网类型信号，由待测模块输出LED状态；data为8位寄存器类型信号，用于在仿真中存储随机数，作为调试参考。

仿真流程如下：首先在初始化阶段，rst_n被拉低保持复位，同时时钟信号clk置为0。经过20ns后释放复位,模块进入正常工作状态。随后,testbench通过always语句产生周期为20ns的时钟信号，即50MHz。此时led_test模块被实例化并开始运行，参数CNT_MAX被设置为10。在运行过程中，testbench在每个时钟上升沿产生新的随机数data，当复位有效时data清零，否则赋值为$random取模256的结果，并打印输出。同时，使用$monitor实时监控led信号的变化。当仿真运行至2000ns时，打印“I am stop”提示并调用$stop命令终止仿真。

在调试过程中，需要重点关注以下几个方面。首先检查复位逻辑，确认led_test在rst_n拉低时能够正确复位。其次观察LED输出，确认其随内部逻辑正常变化。再次检查随机数data的变化情况，确保testbench在每个时钟周期都能产生不同的值，验证输入驱动的多样性。最后需要关注仿真时间是否满足需求，本例默认运行2000ns，可根据测试范围进行调整。

2.5.Modelsim 的使用

该部分主要介绍Modelsim的基本使用方法。

当我们的设计文件没有使用到任何平台的IP核时，我们可以直接打开Modelsim新建工程，然后进行仿真，具体步骤如下所示：

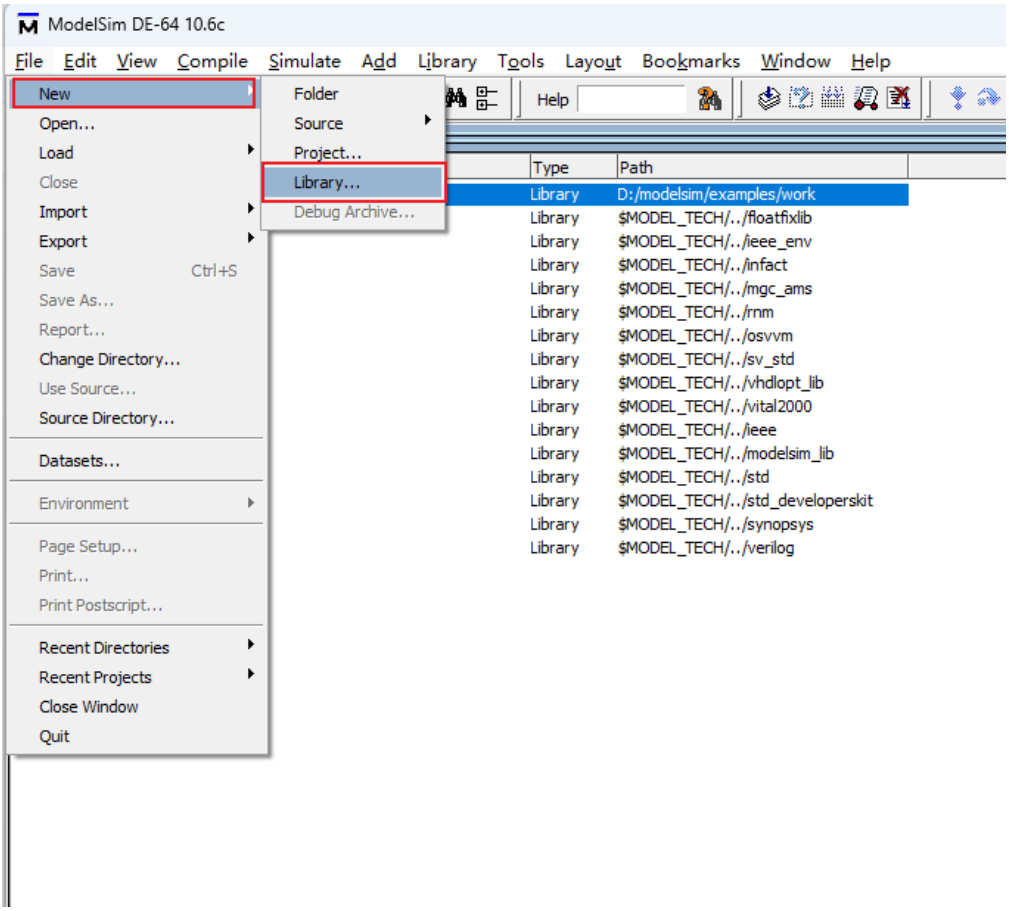


图 2.5-1

点击左上角File->New->Library，新建一个工作库，一般取名为work，因为Modelsim运行时都会在这个work下面工作，所以第一次运行Modelsim我们需要新建一个叫work的库，如果打开发现已经有work的工作库时，则不用新建。

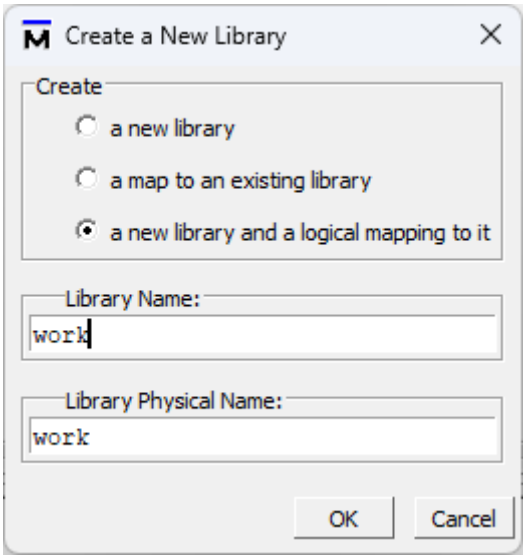


图 2.5-2

输入work，点击OK即可。新建完成后就可以看到有个work的库在Modelsim里面。接下来新建工程。

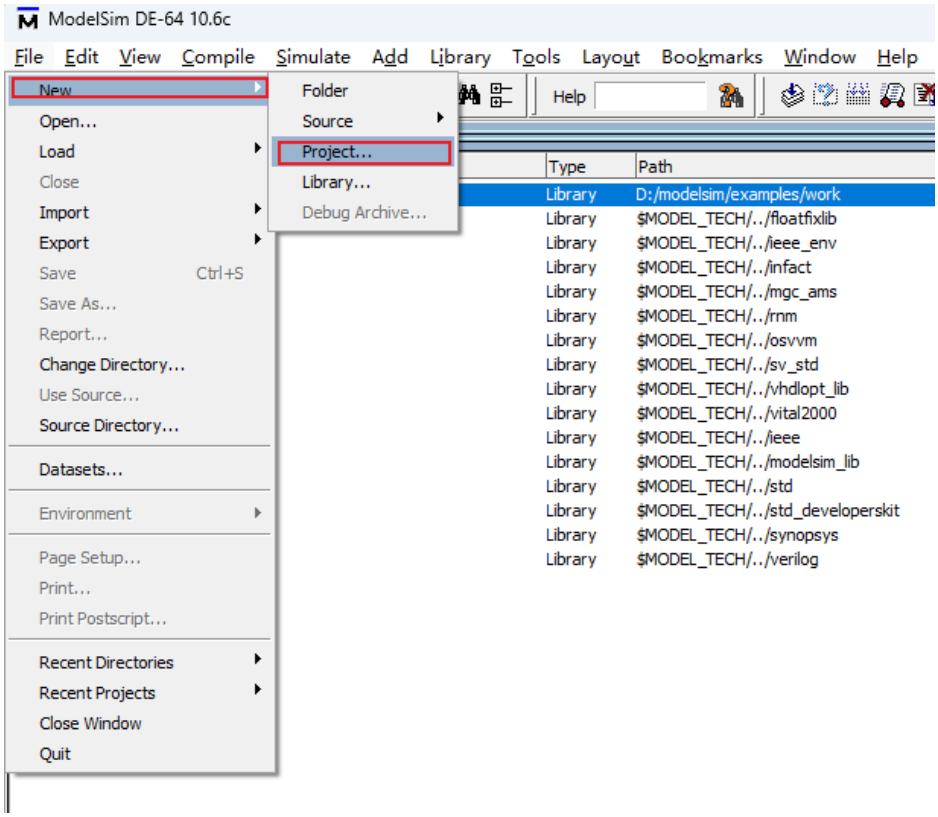


图 2.5-3

左上角File->New->Project，新建工程。

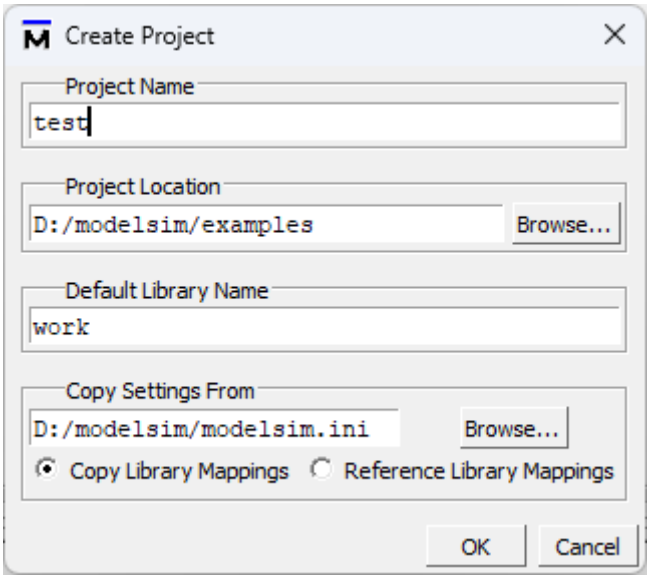


图 2.5-4

工程名注意不要出现中文，其余保持默认即可，可以看到Default Library Name其名字默认指向work。

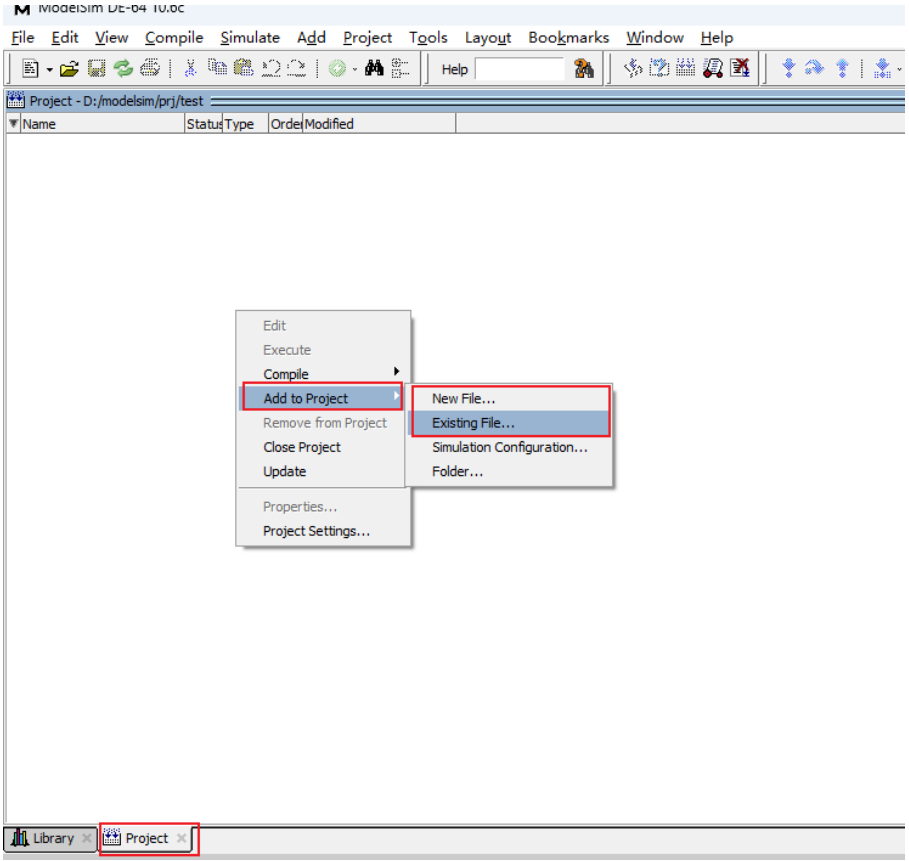


图 2.5-5

新建完后可以看到下方多了一个叫Project的选项卡，鼠标右键该界面空白部分，选择Add to Project->Existing File，或者Add to Project->New File。添加我们要仿真的文件，这里用一个比较简单的来演示。

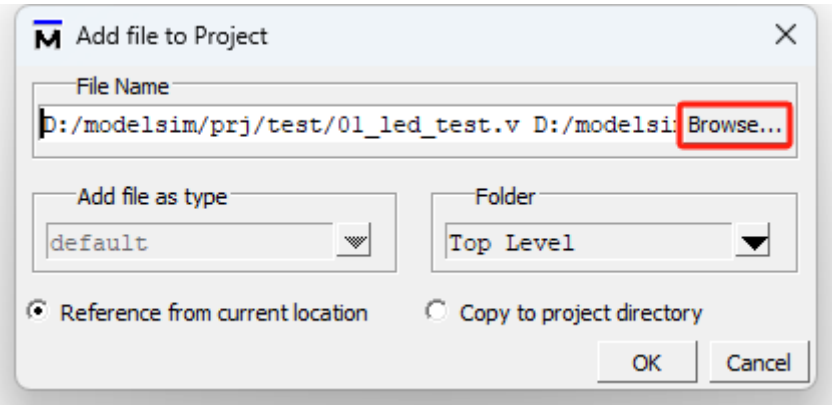


图 2.5-6

点击Browse，添加要仿真的文件。

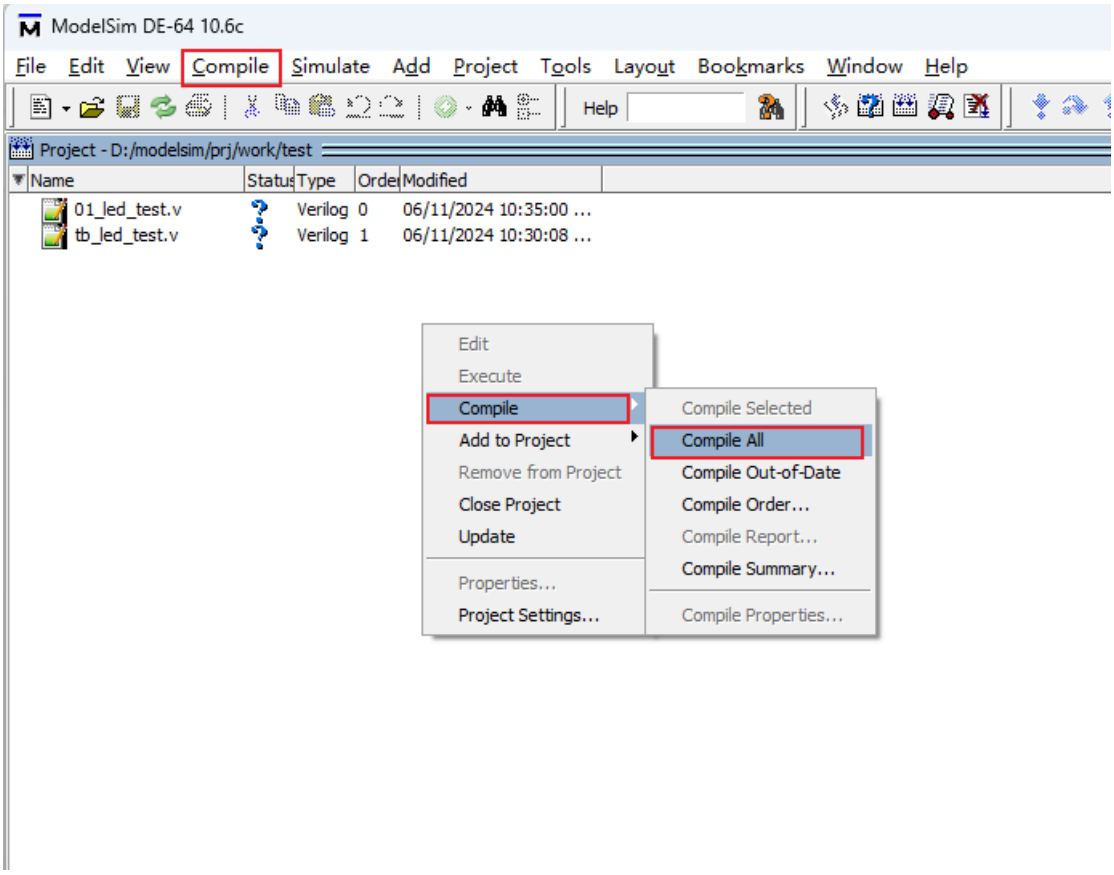
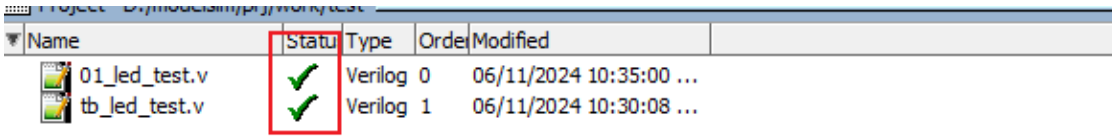


图 2.5-7

选择上方Compile或者鼠标右键空白部分，选择Compile->Compile，该步骤主要对verilog文件进行编译，检查是否有语法错误等。



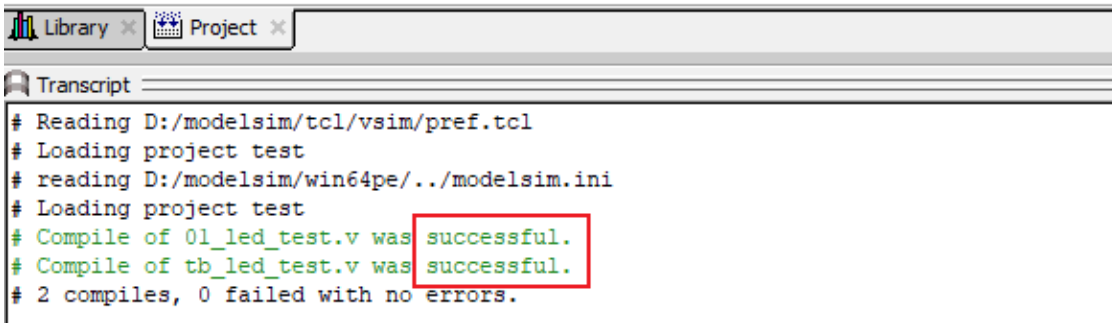


图 2.5-8

当看到Status是个绿色的√，或者下方打印输出区间没有任何errors，显示successful，表示我们的文件编译通过，可以进行下一步操作了。

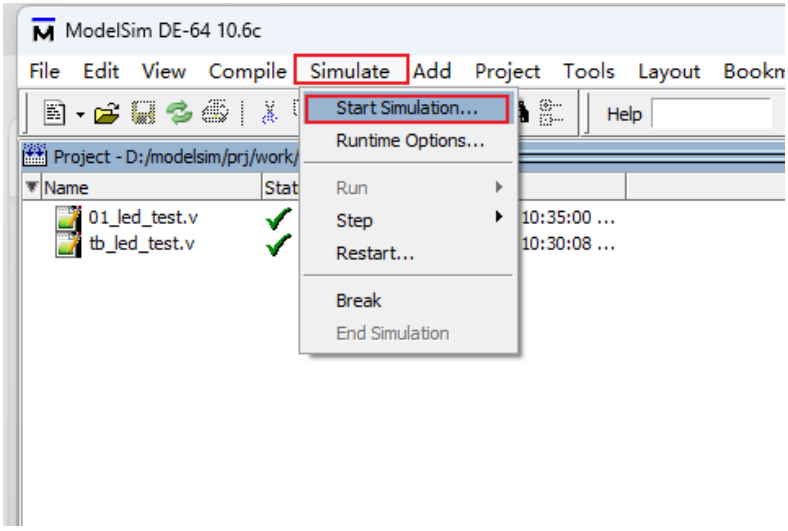


图 2.5-9

选择上方Simulate->Start Simulation，之后会弹出如图 2.5-10所示的界面，把work展开，选择我们的testbench文件，可以看到Design Unit显示的是我们的testbench文件就没问题了。然后点击OK，开始仿真。

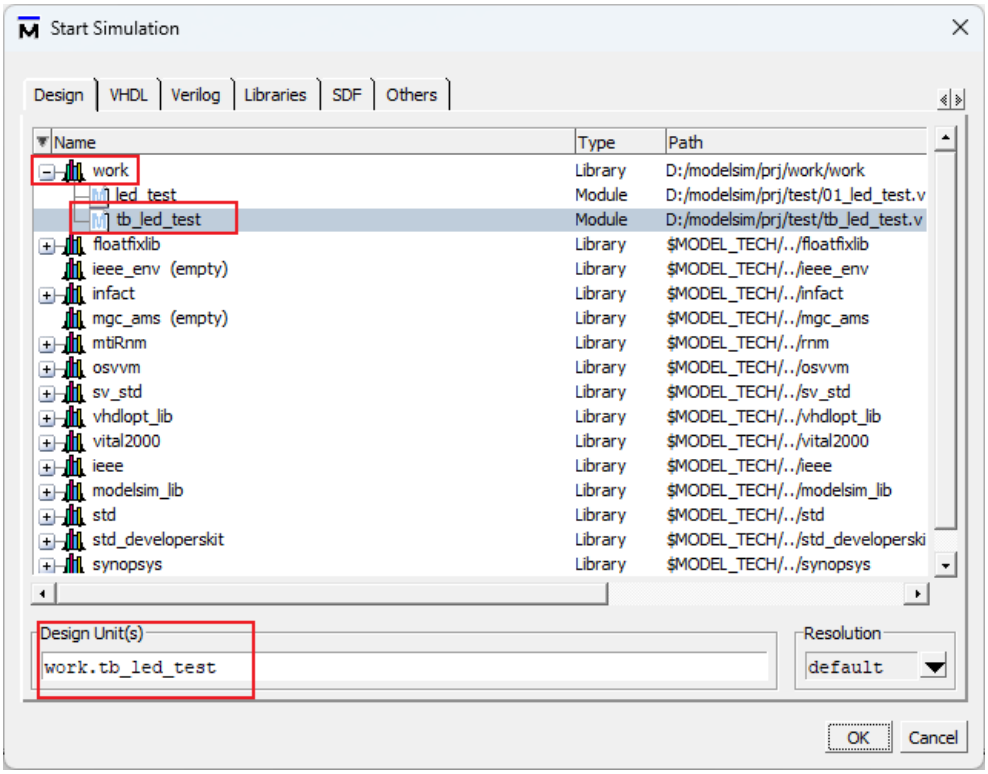


图 2.5-10

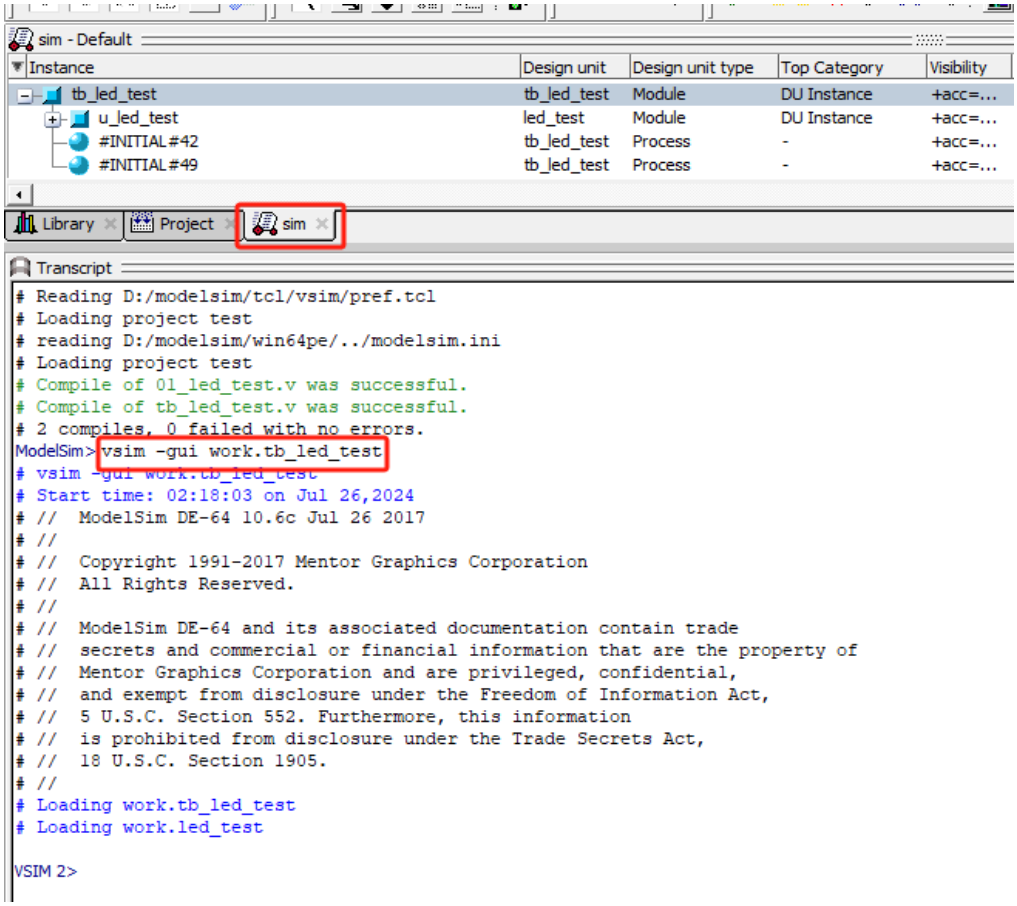


图 2.5-11

可以看到会弹出一个新的选项卡叫”sim”，然后看红框部分，当点击OK后，实际上Modelsim自动输入一句命令vsim -gui work.tb_led_test。这其实和后面我们的do文件编写是有联系的，do文件的编写实际上就是在写这些命令，这里我们先铺垫一下。

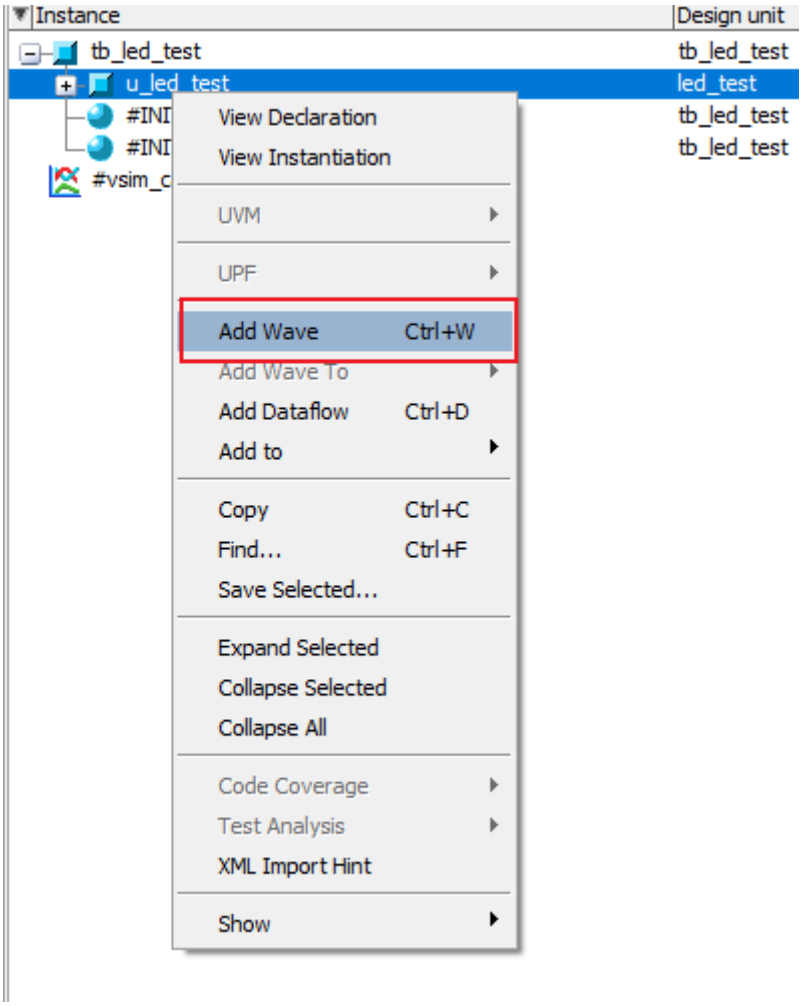


图 2.5-12

接下来添加我们要观察的信号，这里我是直接右键u_led_test这个模块，然后选择Add Wave或者crtl+w，即可将该模块的全部信号都加入到波形窗口中。

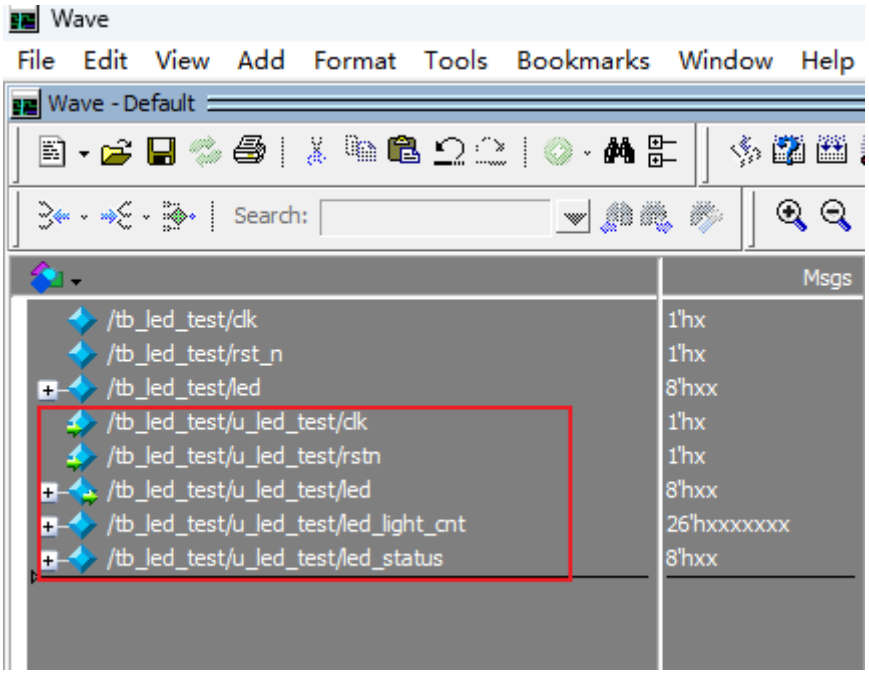


图 2.5-13

可以看到，波形窗口已经添加该模块的全部信号，之后我们按下快捷建，crtl+a全选全部信号，crtl+g，对信号进行分组，该分组是按照不同模块进行分组，crtl+h消除信号名称的前缀，如图 2.5-14所示：

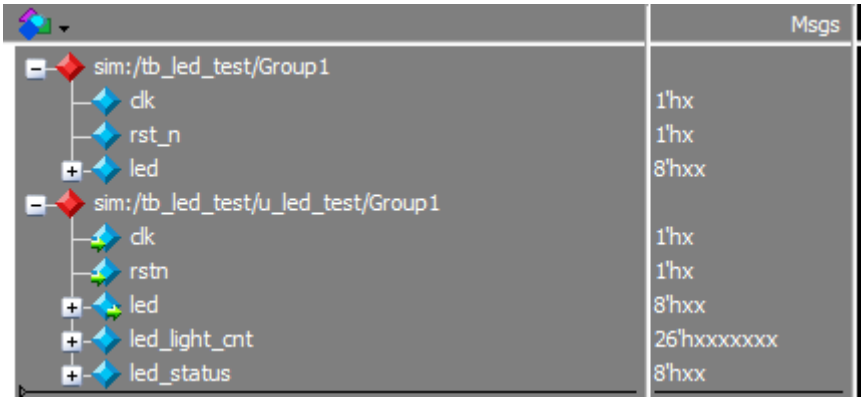


图 2.5-14

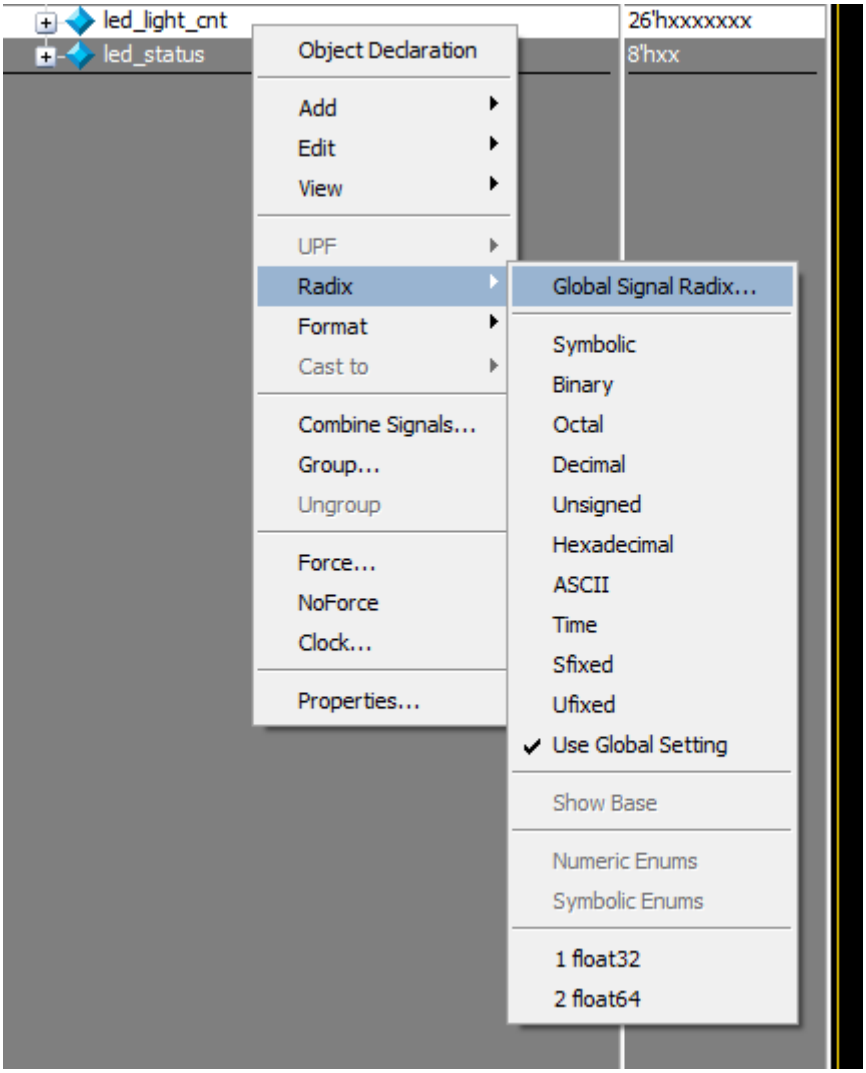


图 2.5-15

鼠标右键信号，Radix可以修改该信号显示的格式，比如二进制显示，16进制显示等。Properties可以修改该信号波形显示的颜色，这两个是比较常用的。接下来开始来运行我们的仿真。

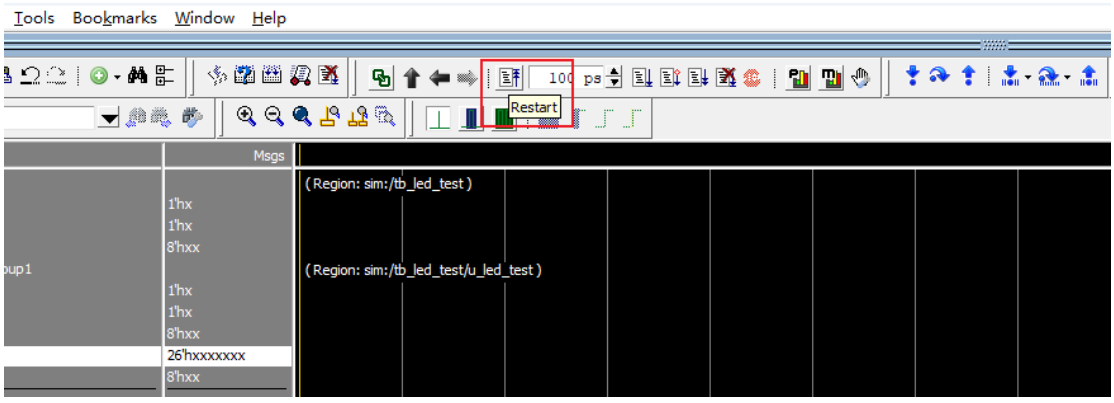


图 2.5-16

点击上方这个地方，对信号全部进行Restart复位。

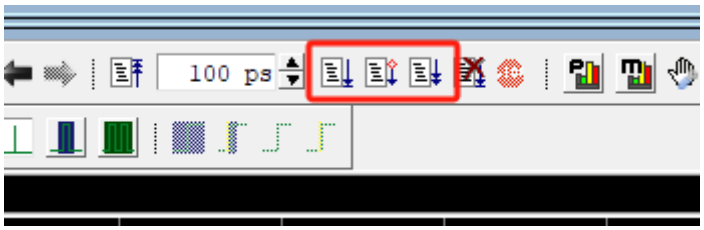


图 2.5-17

红框旁边的100ps是一次仿真运行的时间，红框内从左往右看，第一个是表示运行一次仿真，其时间为100ps，100ps并不是固定的，我们可以修改为1ms,100us等。第二个基本比较少用。第三个是让仿真不断运行，直到用户点击停止为止，如图 2.5-18所示：

图 2.5-18

按下后，当用户点击stop，仿真才会停止。

图 2.5-19

图 2.5-19为操作后显示出来的波形。

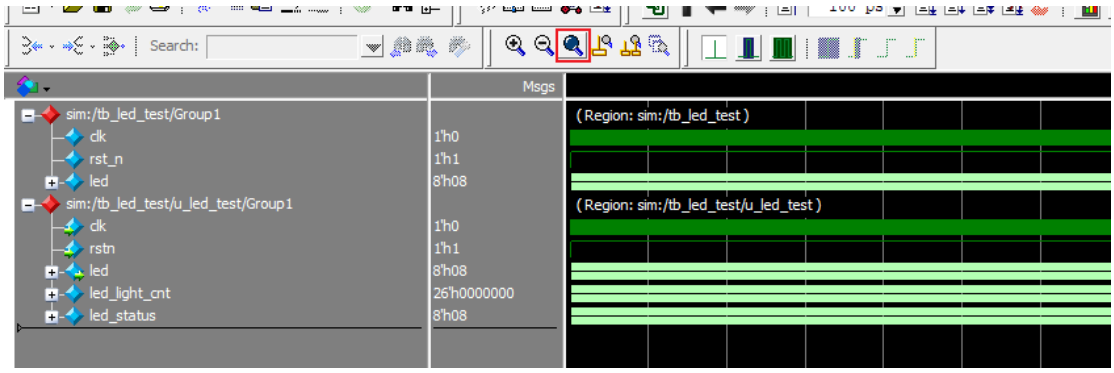


图 2.5-20

点击红框部分那个按钮，将缩小波形。我们也可以按住crtl键，然后鼠标滚轮上下，可以对波形进行缩放。到这里，基本的使用方法就结束了，更多操作大家可以去看视频操作，或者网上百度，或者自己摸索一下。

再铺垫一下，完成这些操作后，我们回去打印输出区间观察一下

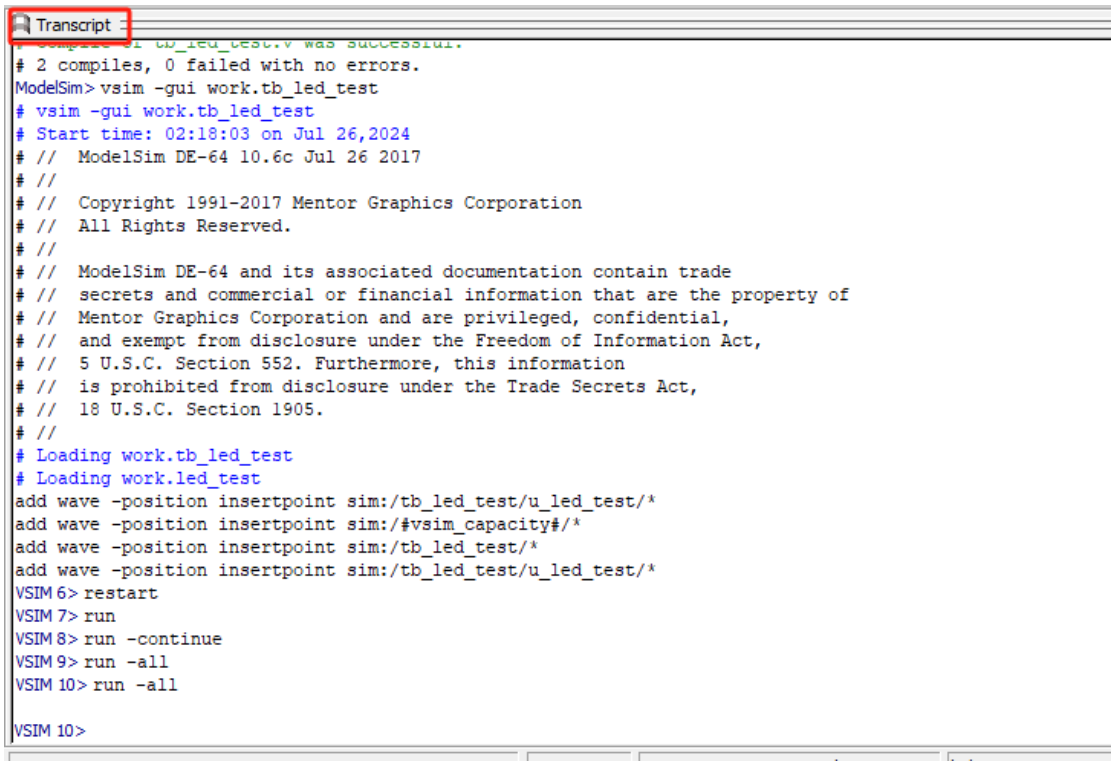


图 2.5-21

可以看当我们添加波形时，Modelsim自动执行了一句add wave -position xxxxxxx的命令，执行了restart，也就是复位，run就是运行仿真，这些都和后续do文件的编写息息相关。所以其

本质就是编写这些命令，我们就不需要用鼠标去点每个功能，每次我们只需要运行do文件就可以完成全部操作，大大提高我们的效率。

如果大家不小心把某些选项卡关了，可以在上方View选择要查看的窗口，如图 2.5-22所示：

比如Library前面有个√，就是显示Library选项卡的意思。大家可以在这里找找需要显示的界面。

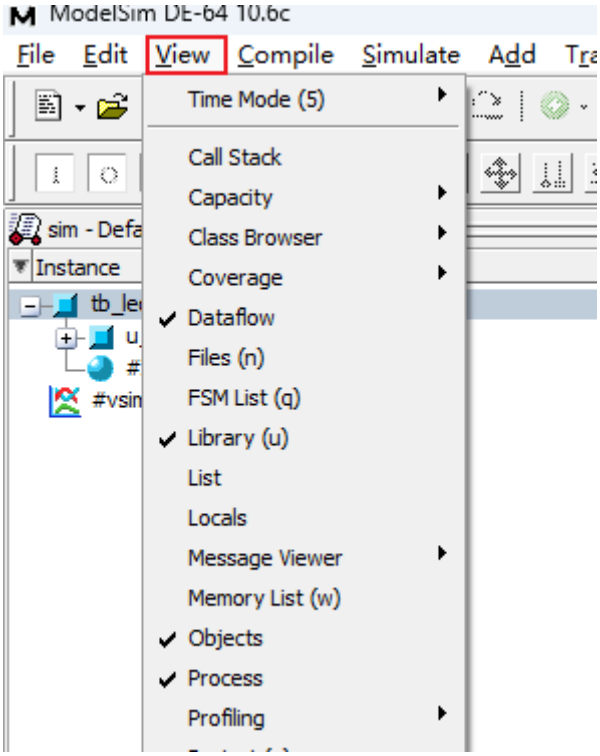


图 2.5-22

2. 6. 文件的编写

2.6.1.基本命令介绍

前文其实也有提到，Modelsim实际上是通过输入命令来执行相应功能的，比如add wave xxxxx，就是把信号添加到波形窗口。那接下来就是教大家如何使用这些命令来提高我们的开发效率。

首先先介绍常用的命令。

vlib:该命令为创建一个目录。例如 vlib work。即在当前路径下创建一个名字叫work的文件夹。

vmap: 映射逻辑库到物理目录。其格式为vmap work work 第一个work逻辑库名称，第二个work是表示在PC里实际的库文件的路径。

注意：前面所说的通过File->New->Library的方法建立了一个work的库，其实就是运行了vlib和vmap，具体可以看教程视频讲解。本质就是vlib work vmap work work。

vlog:该命令用来编译verilog源码。例如vlog -work work ./src/test.v 第一个work表示文件夹的名称 、第二个work表示modelsim中library的库的名称、第三个就是要编译的文件的路径。

vsim:表示启动仿真。

add wave:表示添加波形到波形窗口(add wave -divider 会添加分割线)。

view wave:打开波形窗口。

view structure:打开结构窗口。

view signals:打开信号窗口。

restart:重新仿真，复位仿真时间，并清空之前的仿真数据。(如果修改了verilog文件 需要重新编译再仿真才行，restart只是在当前这个仿真下重新开始仿真而已)。

run x:运行x时间。例如run 1ms run1ns run 1us run 250ms 均可。

quit -sim:退出仿真。

quit:退出Modelsim。（关闭整个软件）

2.6.2.文件示例

如果从0开始写，相信是比较陌生的，其实当我们使用紫光联合仿真的时候，他会在sim的文件夹下生成一个后缀为tcl的脚本，每次运行联合仿真，实际就是打开Modelsim然后运行该tcl脚本，具体路径都在工程目录下的sim文件夹下，如图 2-1所示：

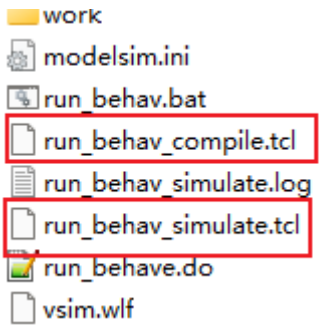


图 2-1

主要是运行run_behav_compile.tcl和run_behav_simulate.tcl这两个文件。我们可以打开来参考一下。

```
1.  vlib  work
2.  vmap  work ./work
3.  vmap  usim "D:/modelsim/pg_sim_lib/usim"
```

```
4.  vmap  adc_e2 "D:/modelsim/pg_sim_lib/adc_e2"
5.  vmap  ddc_e2 "D:/modelsim/pg_sim_lib/ddc_e2"
6.  vmap  dll_e2 "D:/modelsim/pg_sim_lib/dll_e2"
7.  vmap  hsstlp_lane "D:/modelsim/pg_sim_lib/hsstlp_lane"
8.  vmap  hsstlp_pll "D:/modelsim/pg_sim_lib/hsstlp_pll"
9.  vmap  iolhr_dft "D:/modelsim/pg_sim_lib/iolhr_dft"
10. vmap  ipal_e1 "D:/modelsim/pg_sim_lib/ipal_e1"
11. vmap  ipal_e2 "D:/modelsim/pg_sim_lib/ipal_e2"
12. vmap  iserdes_e2 "D:/modelsim/pg_sim_lib/iserdes_e2"
13. vmap  oserdes_e2 "D:/modelsim/pg_sim_lib/oserdes_e2"
14. vmap  pciegen2 "D:/modelsim/pg_sim_lib/pciegen2"
15. vlog -work work   \
16. "D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desktop/01_led_test.v" \
17. "D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desktop/tb_led_test.v" \
18. "D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/source/Desktop/01_led_test.v"
```

以上是run_behav_compile.tcl的内容，大家可以结合视频教程一起分析一下，该文件主要完成工作区间的建立和一些库的映射以及对代码的编译。

vlib:该命令创建一个文件夹。例如 vlib work。

vmap:映射逻辑库到物理目录。其格式为vmap work work 第一个work逻辑库名称，第二个work是表示在PC里实际的库文件的路径。

vlog:该命令用来编译verilog源码。例如vlog -work work ./src/test.v

第一个work表示文件夹的名称。第二个work表示Modelsim中library的库的名称。第三个就是要编译的文件的路径。

所以大家其实可以参考demo的脚本来编写我们的do文件，我们的do文件本质上也是写这些命令，只不过后缀不一样，但其运行方法是一致的，均为do+空格+文件名。所以到此，大家应该都知道我们的do文件是怎么去编写了，其实就是把这些Modelisim的运行指令，写成一个脚本，然后用do指令直接完成我们想要的所有操作，可以大大提高我们的效率。在展示如何使用Modelsim的时候也介绍了，每一步操作实际上都是软件工具自动帮我们输入命令，现在就是把这些命令给拿出来。接下来我们看run_behav_simulate.tcl的内容。

```
1.  vsim  -novopt  -L work -L usim -L adc_e2 -L ddc_e2 -L dll_e2 -L hsstlp_lane -L hsstlp_pll -L iolhr_dft -L ipal_e1 -L ipal_e2 -L iserdes_e2 -L oserdes_e2 -L pciegen2 tb_led_test usim.GTP_GRS
2.  add wave *
3.  view wave
4.  view structure
5.  view signals
6.  run 1000ns
7.  vsim  -novopt  -L work -L usim -L adc_e2 -L ddc_e2 -L dll_e2 -L hsstlp_lane -L hsstlp_pll -L iolhr_dft -L ipal_e1 -L ipal_e2 -L iserdes_e2 -L oserdes_e2 -L pciegen2 tb_led_test usim.GTP_GRS
8.  add wave *
9.view wave
10.view structure
11.view signals
12.run 1000ns
13.
```

vsim:表示启动仿真。vsim -L +逻辑库的名字。

add wave:表示添加波形到波形窗口。(add wave -divider 会添加分割线)

view wave:打开波形窗口。

view structure:打开结构窗口。

view signals:打开信号窗口。

run x:运行x时间。例如run 1ms run1ns run 1us run 1s run 250ms 均可。

再顺带介绍一下一些常用的。

restart:重新仿真，复位仿真时间，并清空之前的仿真数据。(如果修改了verilog文件 需要重新运行do文件才生效，restart只是在当前这个仿真下重新开始仿真而已)

quit -sim:退出仿真。

quit:退出Modelsim。

该脚本主要是完成仿真，以及一些仿真完成后的操作，比如添加波形，观察波形，设置运行时间。

所以，其实我们可以把这两个文件合起来，变成一个文件，做成我们自己的do文件就行了，如此，以后修改代码重新仿真都不需要去PDS软件里面去点联合仿真，我们直接在Modelsim里面直接do就行了。合并后的do文件如下所示：

```
cd D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/sim/behav
vlib work
vmap work ./work
vmap usim "D:/modelsim/pg_sim_lib/usim"
vmap adc_e2 "D:/modelsim/pg_sim_lib/adc_e2"
vmap ddc_e2 "D:/modelsim/pg_sim_lib/ddc_e2"
vmap dll_e2 "D:/modelsim/pg_sim_lib/dll_e2"
vmap hsstlp_lane "D:/modelsim/pg_sim_lib/hsstlp_lane"
vmap hsstlp_pll "D:/modelsim/pg_sim_lib/hsstlp_pll"
vmap iolhr_dft "D:/modelsim/pg_sim_lib/iolhr_dft"
vmap ipal_e1 "D:/modelsim/pg_sim_lib/ipal_e1"
vmap ipal_e2 "D:/modelsim/pg_sim_lib/ipal_e2"
vmap iserdes_e2 "D:/modelsim/pg_sim_lib/iserdes_e2"
vmap oserdes_e2 "D:/modelsim/pg_sim_lib/oserdes_e2"
vmap pciegen2 "D:/modelsim/pg_sim_lib/pciegen2"
vlog -work work \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desktop/01_led_test.v" \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desktop/tb_led_test.v" \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/source/Desktop/01_led_test.v"

vsim -novopt -L work -L usim -L adc_e2 -L ddc_e2 -L dll_e2 -L hsstlp_lane -L hsstlp_pll -L iolhr_dft -L ipal_e1 -L ipal_e2 -L iserdes_e2 -L oserdes_e2 -L pciegen2 tb_led_test usim.GTP_GRS
add wave *
add wave -position insertpoint sim:/tb_led_test/u_led_test/*
view wave
```

```
view structure
view signals

restart
run 1000ns

```

可以看到，基本上就是把两个文件给合并起来，然后多添加了restart语句。至于添加波形的语句add wave -position insertpoint sim:/tb_led_test/u_led_test/* ，如果大家不熟悉这样的格式，可以直接在Modelsim里面手动添加，然后看其打印区间，输出的指令格式，复制下来就行了。一开始，大家不熟悉的话可以这么操作，等熟悉了后，就可以完全编写了，因为使用了紫光的联合仿真，所以中间会用vmap映射很多的紫光的仿真库。包括vsim也调用了很多紫光相关的库。所以，如果大家并没有用到紫光的IP核或者原语等，只是单纯的验证逻辑的话，其实没有这么麻烦。与紫光的仿真库有关的都可以删了，所以主要就是一个vlib work，然后vmap work work，然后vlog我们要仿真的文件的路径，注意需要写好testbench。然后vsim，然后添加要查看的波形，然后restart，然后run即可。

3.Pango 与 Modelsim 的联合仿真

3. 1. 实验简介

实验目的：完成PDS软件和Modelsim的联合仿真设置，所有版本设置方法基本一致，这里以PDS2022.2版本为例。

实验环境：

Window11

PDS2022.2

Modelsim10.6rc

硬件环境：

暂无

3. 2. 实验原理

编写完成Testbench文件后，在PDS设置好Modelsim的路径，即可启动联合仿真。

3.2.1.编译仿真库

首先打开PDS软件，可以不用打开工程，具体图 3.2-1所示：

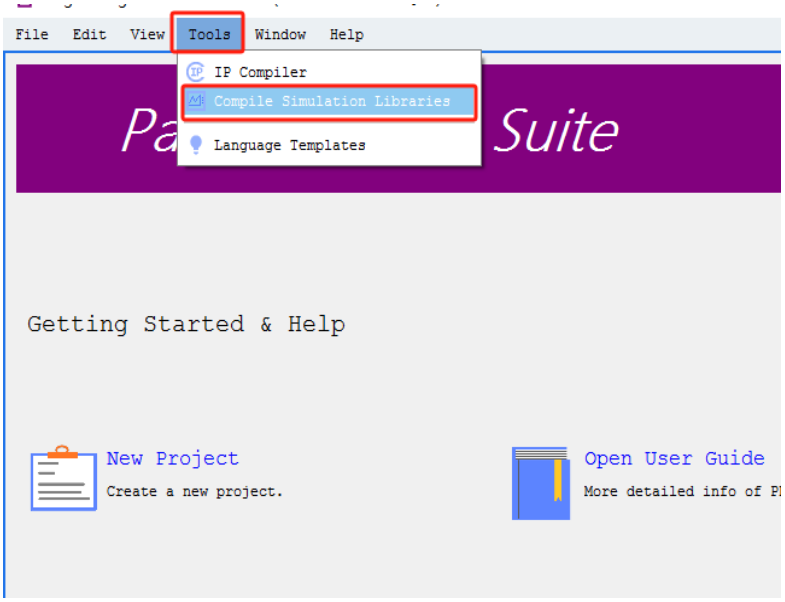


图 3.2-1

不管是否打开工程，均可以在软件上方工具栏中找到Tools->Compile Simulation Libraries;

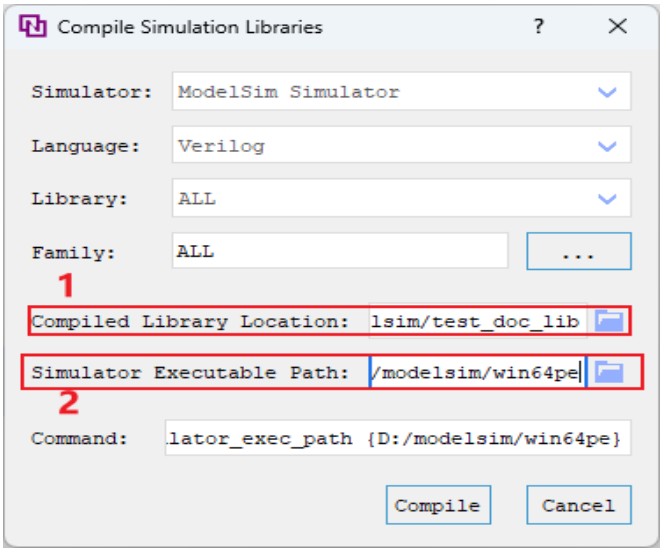


图 3.2-2

打开后可以看到弹出如图 3.2-2所示的界面，其中红框1表示存放生成的仿真库的路径，推荐可以在Modelsim的安装目录下新建一个文件夹来存放，笔者是用pango_sim_lib来表示，通俗易懂。红框2表示Modelsim的启动路径，不同版本其存放的文件夹名字可能不一样，有win64pe/win64/win32等，都是win开头，笔者所用的10.6c为win64pe。之后点击Compile即可，等待编译完成。

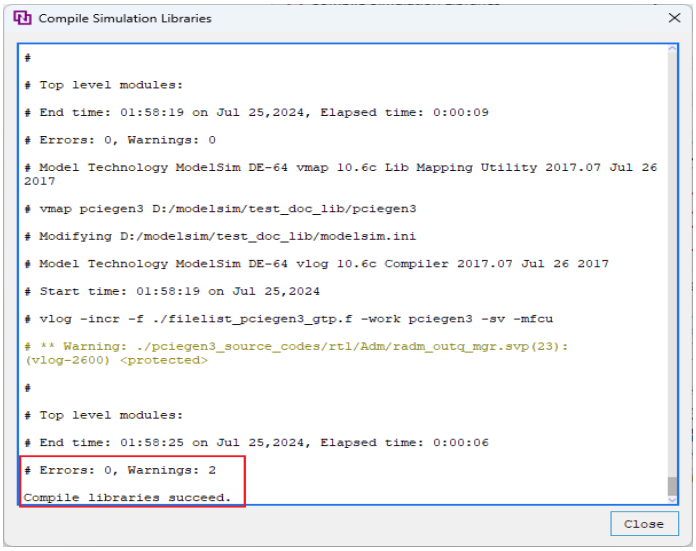


图 3.2-3

当没有任何Errors时(Warnings可以忽略)，表示我们的仿真库已经生成成功了。

3.2.2.设置仿真路径

编译完成仿真库后，我们需要在PDS工程中设置仿真路径，即设置Modelsim的路径以及刚才生成的仿真库的路径。

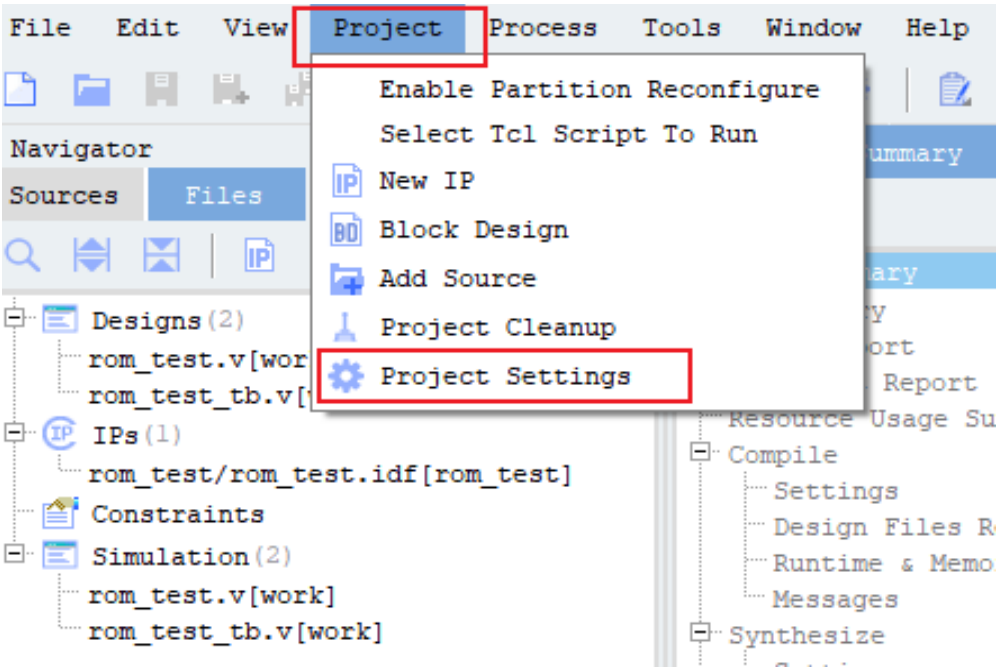


图 3.2-4

打开工程后，选择Project->Project Settings；

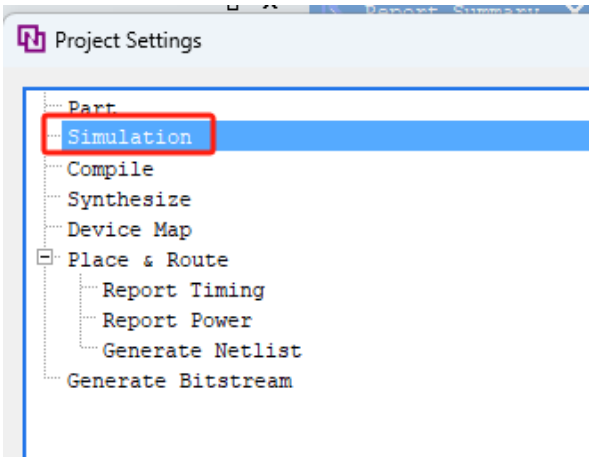


图 3.2-5

选择Simulation选项，准备设置我们的仿真路径。

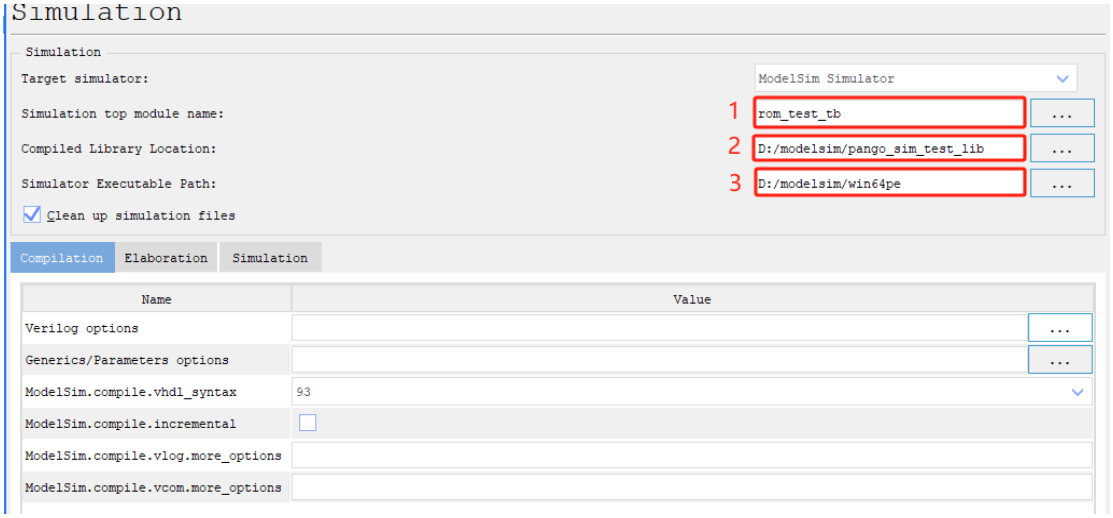


图 3.2-6

接下来开始配置路径，红框1表示我们要仿真的顶层文件，PDS软件会自动识别。红框2选择生成的仿真库的路径。红框3是Modelsim的启动路径，也就是说红框2和红框3的路径和刚才生成仿真库所设置的路径是一模一样的。之后点击OK即可。

3.2.3.启动联合仿真

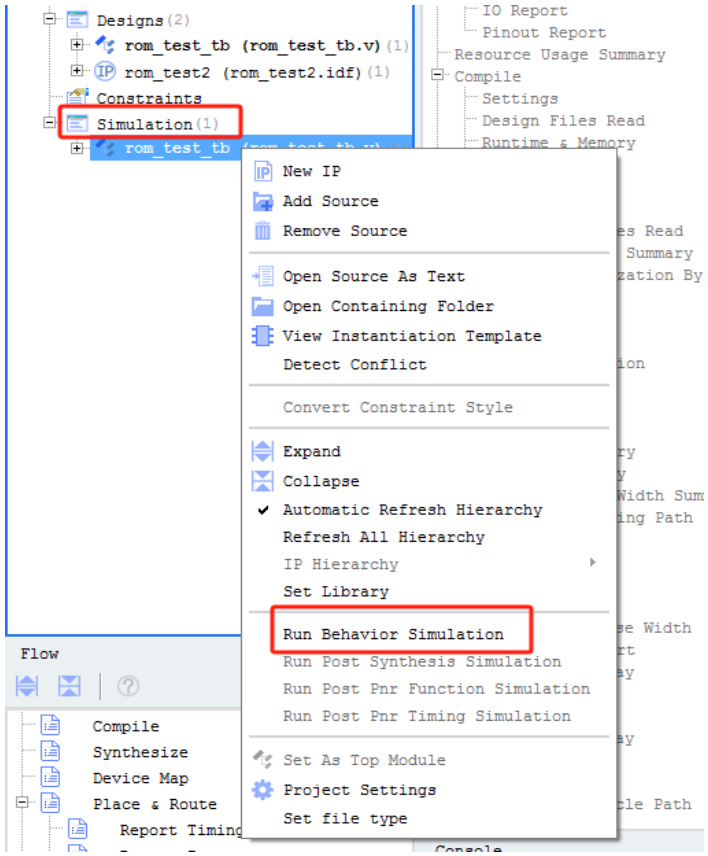
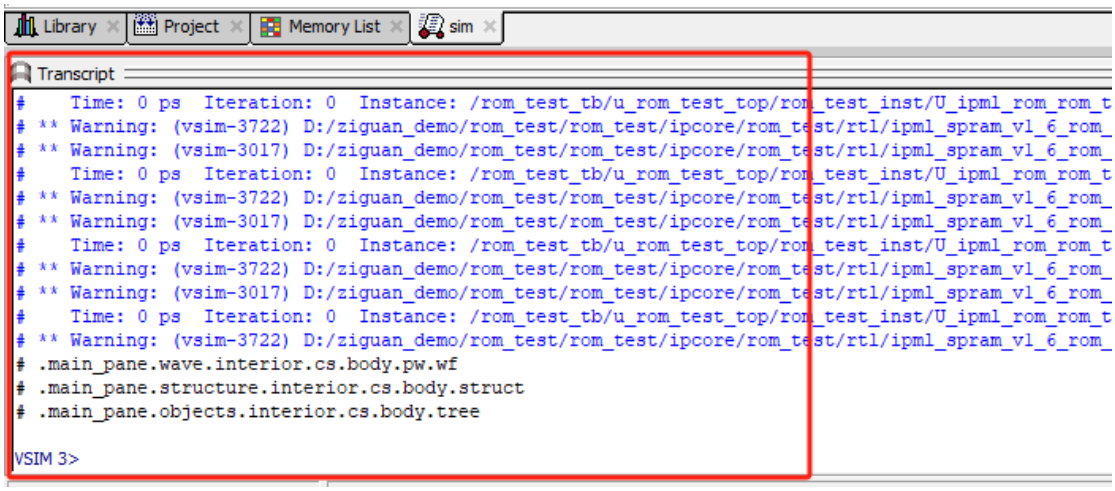


图 3.2-7

接下来在Simulation下，右键仿真的顶层文件，可以看到有四种仿真，我们常用的是第一种行为仿真，可以通过查看仿真波形来验证我们设计的逻辑功能是否正确，该仿真不需要进行任何编译即可直接进行，如果是后面的三种，比如Post Synthesis Simulation则需要综合后才能仿真。接下来点击Run Behavior Simulation，会自动弹出Modelsim的界面。如图 3.2-8所示：



```
#     Time: 0 ps  Iteration: 0  Instance: /rom_test_tb/u_rom_test_top/rom_test_inst/U_ipml_rom_rom_t
# ** Warning: (vsim-3722) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
# ** Warning: (vsim-3017) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
#     Time: 0 ps  Iteration: 0  Instance: /rom_test_tb/u_rom_test_top/rom_test_inst/U_ipml_rom_rom_t
# ** Warning: (vsim-3722) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
# ** Warning: (vsim-3017) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
#     Time: 0 ps  Iteration: 0  Instance: /rom_test_tb/u_rom_test_top/rom_test_inst/U_ipml_rom_rom_t
# ** Warning: (vsim-3722) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
# ** Warning: (vsim-3017) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
#     Time: 0 ps  Iteration: 0  Instance: /rom_test_tb/u_rom_test_top/rom_test_inst/U_ipml_rom_rom_t
# ** Warning: (vsim-3722) D:/ziguan_demo/rom_test/rom_test/ipcore/rom_test/rtl/ipml_spram_v1_6_rom_
# .main_pane.wave.interior.cs.body.pw.wf
# .main_pane.structure.interior.cs.body.struct
# .main_pane.objects.interior.cs.body.tree

VSIM 3>
```

图 3.2-8

打开后Modelsim会自动执行仿真脚本，具体在下个章节会介绍，如果观察到打印区间没有显示任何error，即表示仿真成功，可以开始进行某些操作任何观察波形。

4.紫光同创 IP core 的使用及添加

4. 1. 实验简介

实验目的：

了解PDS软件如何安装IP、使用IP以及查看IP手册

实验环境：

Window11

PDS2022.2

硬件环境：

暂无

4. 2. 实验原理

4.2.1.IP 的安装

PDS软件安装完成之后，PDS自带部分基础IP，其他IP需用户下载IP安装包并安装IP。

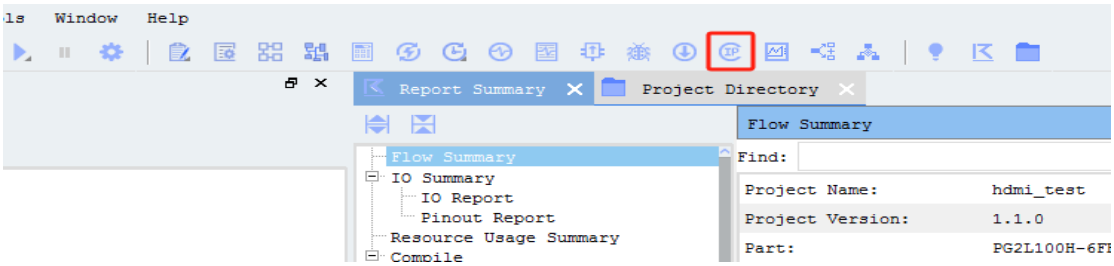


图 4.2-1

打开PDS后，点击图 4.2-1里红框部分的IP图标。

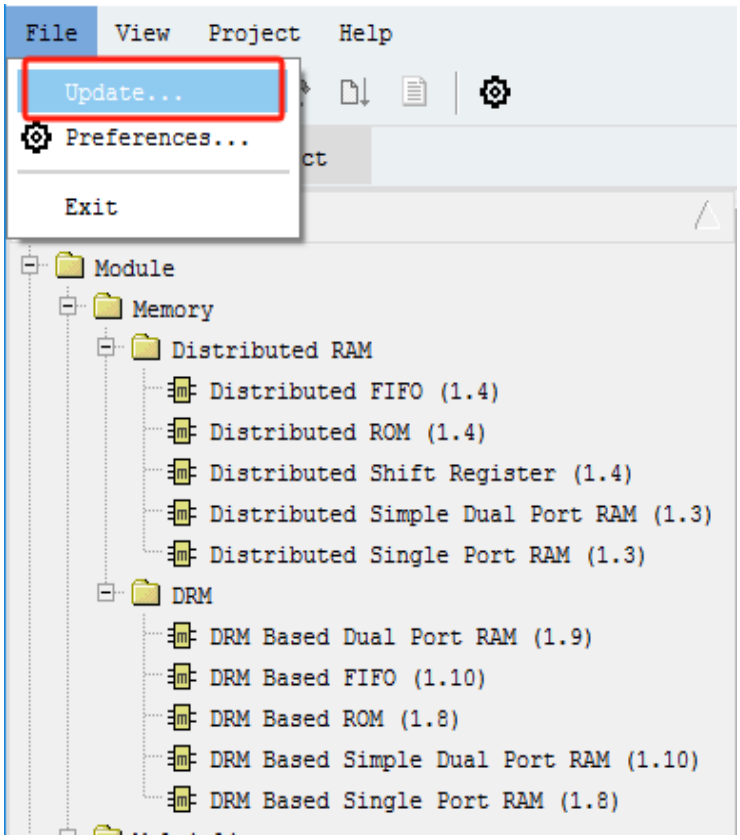


图 4.2-2

之后在弹出的选项卡的左上角点击File->Update...

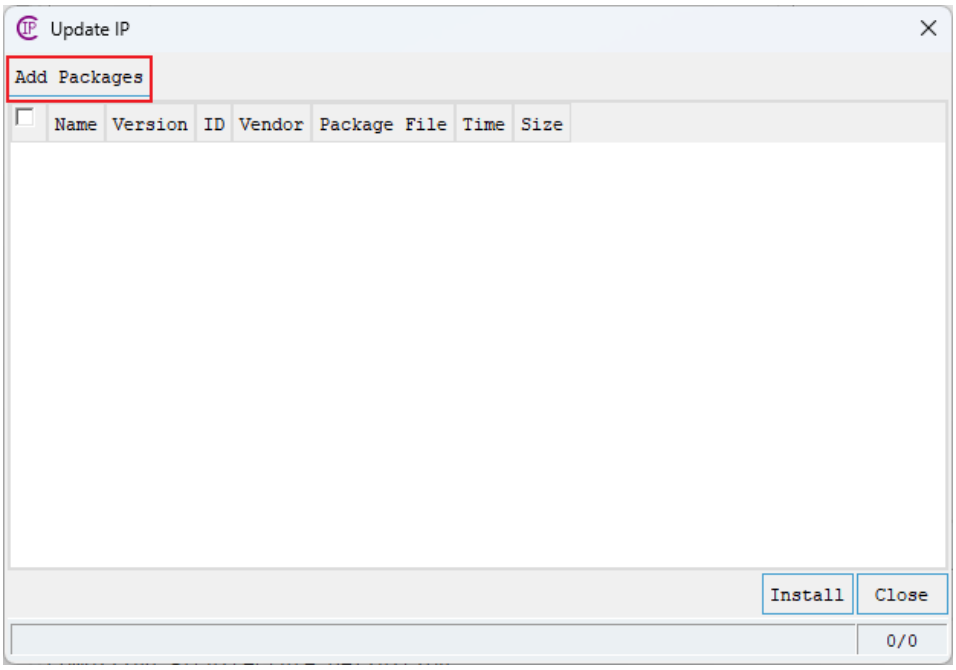


图 4.2-3

点击左上角Add Package。

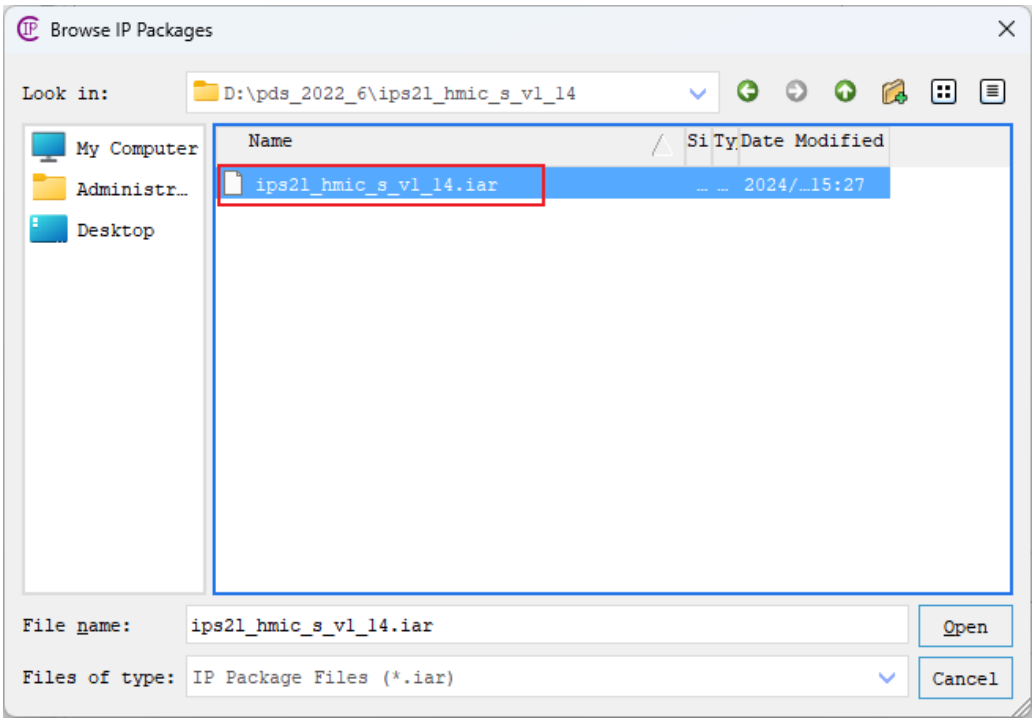


图 4.2-4

如图 4.2-4是DDR3 IP的安装文件，后缀都是.iar。大家选择对应的文件后，点击右下角的Open即可。

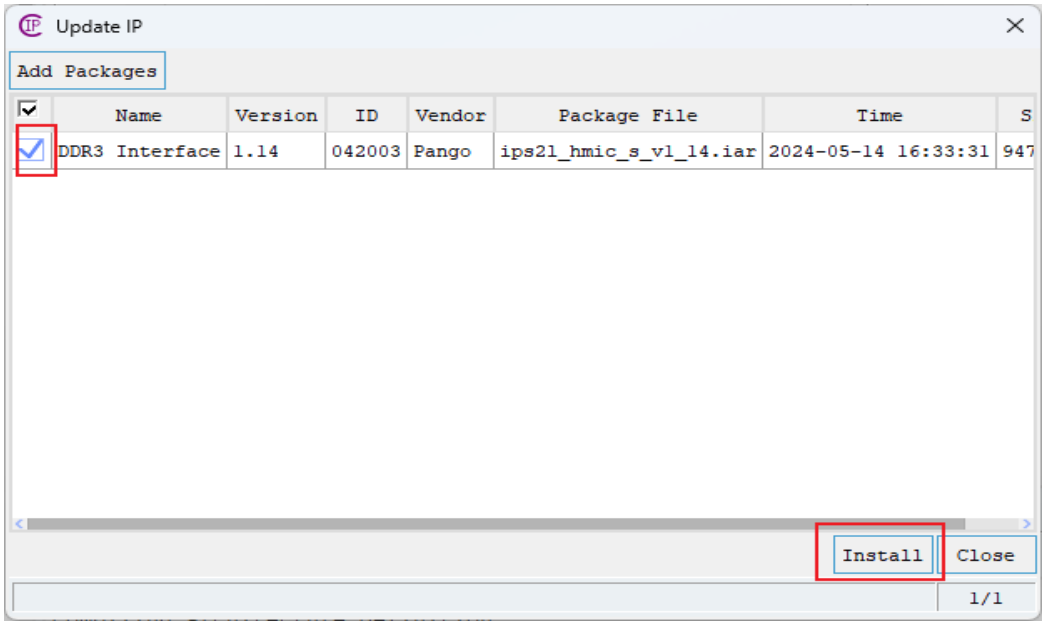


图 4.2-5

之后勾选上前面的√，点击Install即可。

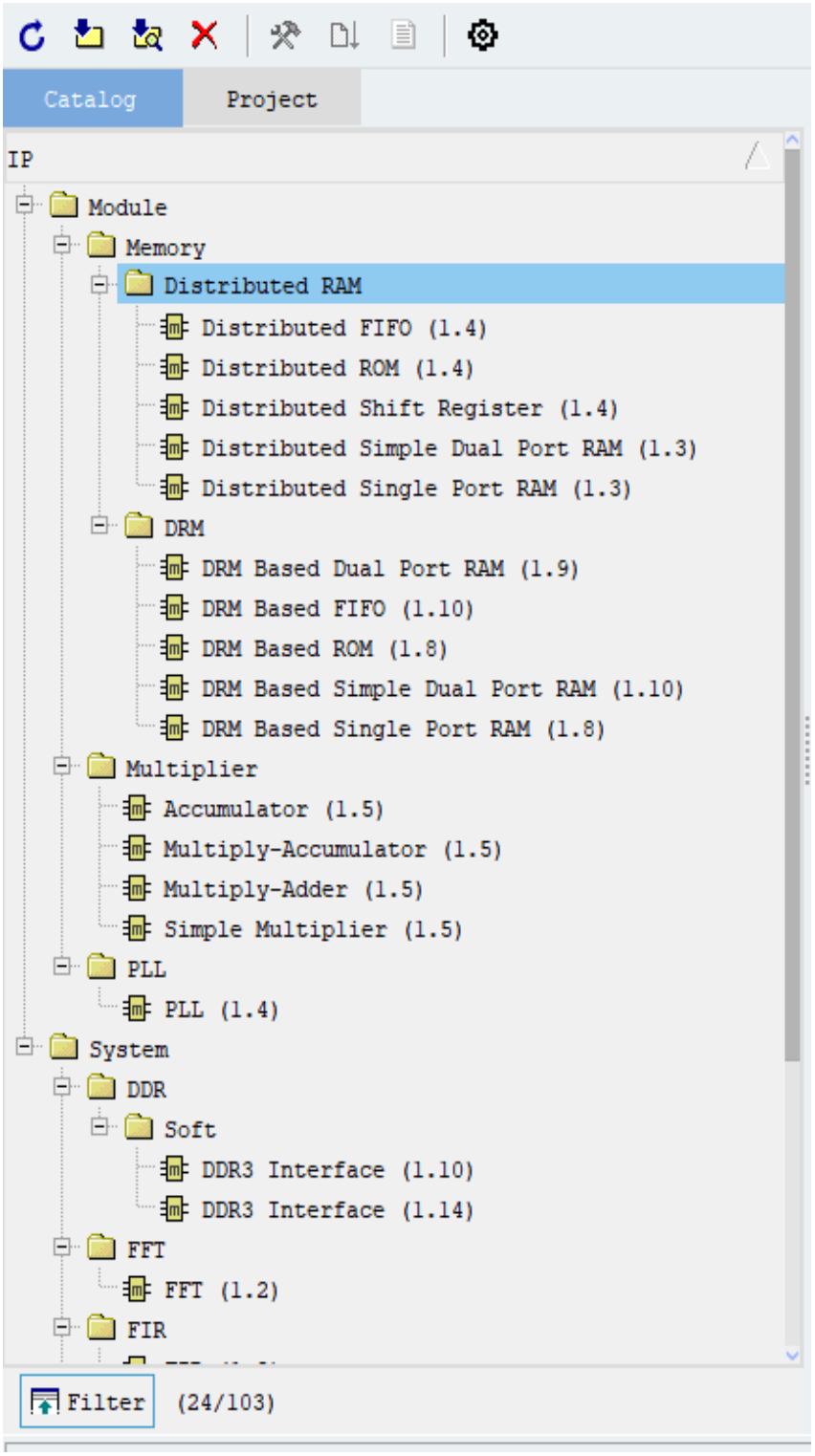


图 4.2-6

之后在左边的界面可以看到刚才安装的IP即可。注意如果发现安装后，弹出了警告，并且左边的界面没有任何变化。那就意味着你安装的IP该系列的器件不支持。因为你的工程可能是LOGOS、LOGOS2、或者Tian2等系列，不同芯片型号所用的IP是不太相同的，所以大家注意这一点。

4.2.2.例化 IP 及查看 IP 手册

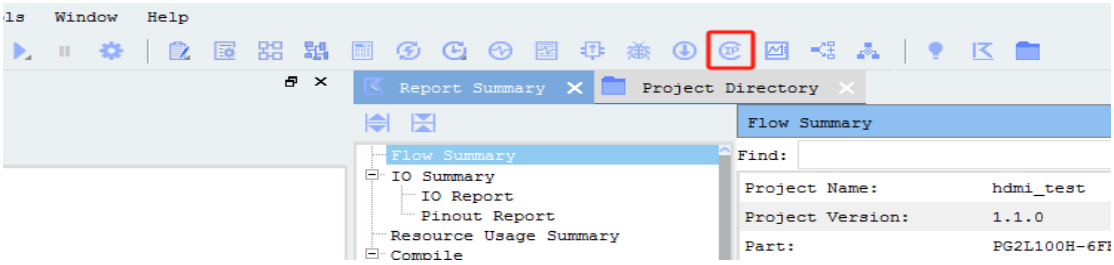


图 4.2-7

继续点击图 4.2-7所示红框的图标。

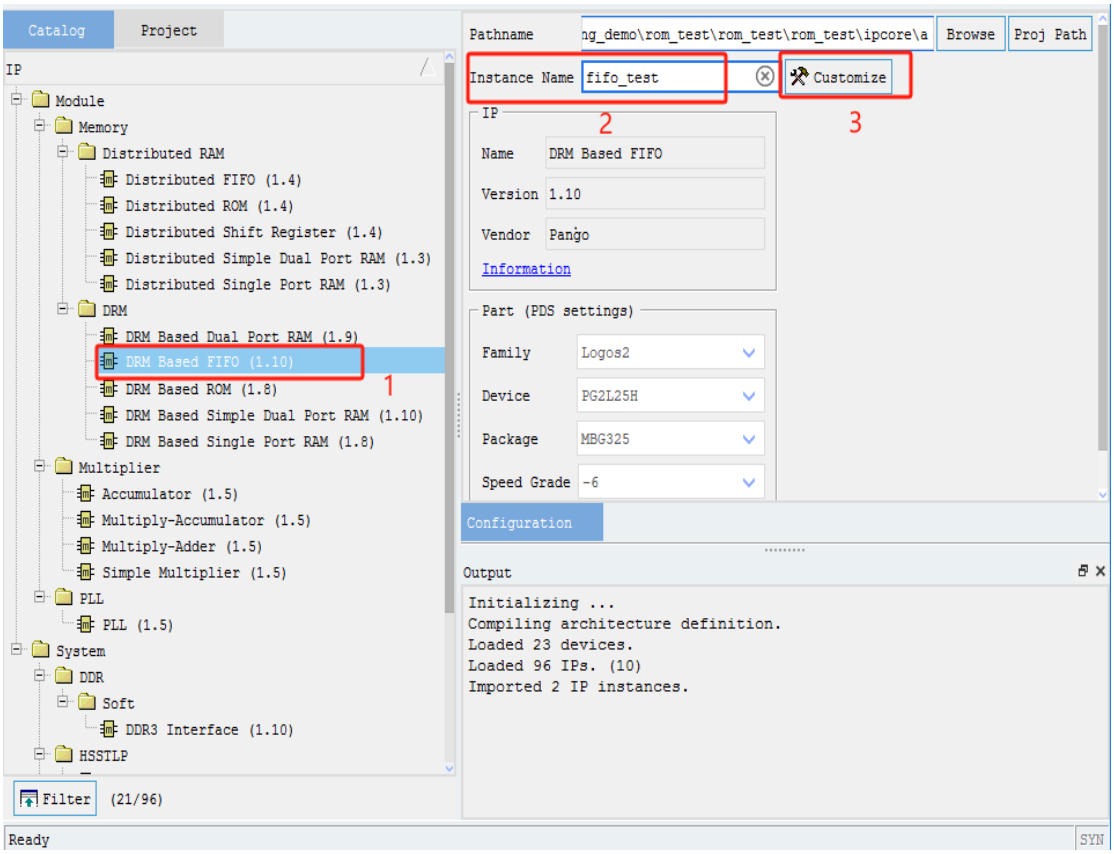


图 4.2-8

选择想要生成的IP，这里以FIFO为例子，即红框1所示。红框2是用来填写生成的IP的名字。点击红框3后即可生成IP，并弹出该IP的配置界面。如图 4.2-9所示：

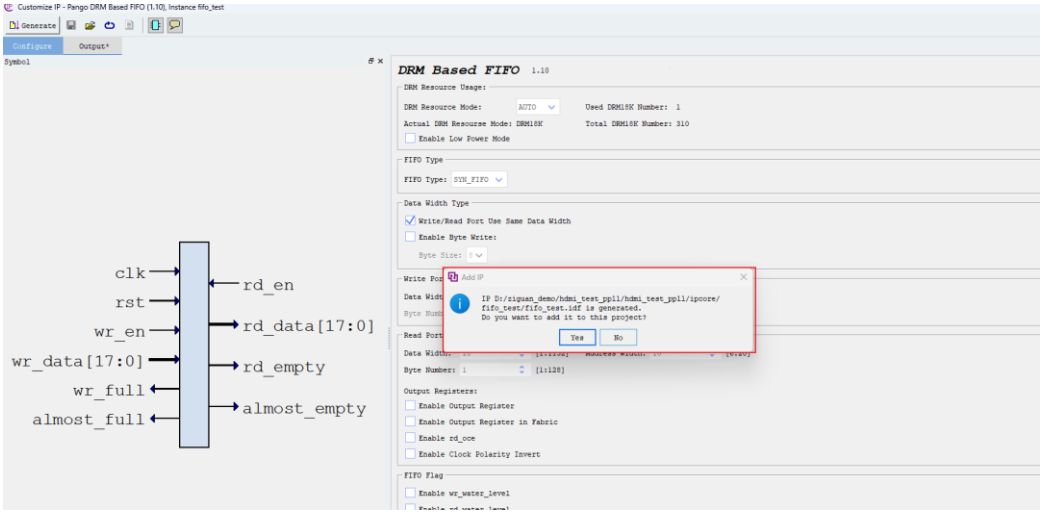


图 4.2-9

其弹出的提示是询问我们是否要把该IP添加到工程中，点击YES就行。如果我们不知道IP如何使用，可以打开官方参考手册查看，如图 4.2-10所示：

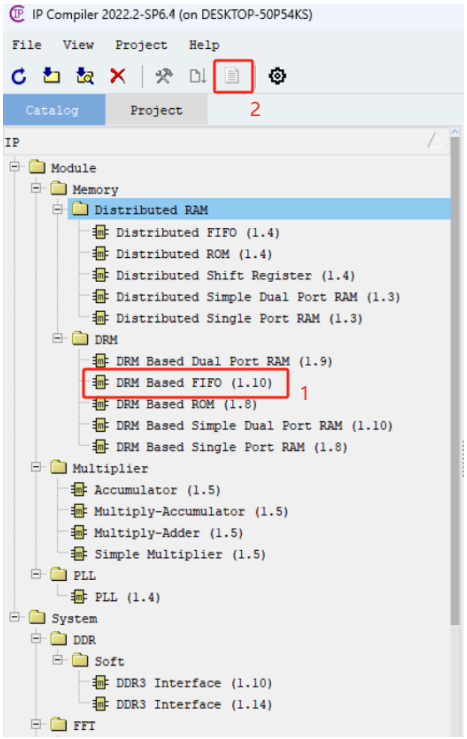


图 4.2-10

选择想要查看的IP，如何点击红框2所示的图标，即可自动弹出官方参考文档。

图 4.2-11

对我们的IP配置完成后，点击左上角红框1处的Generate即可。

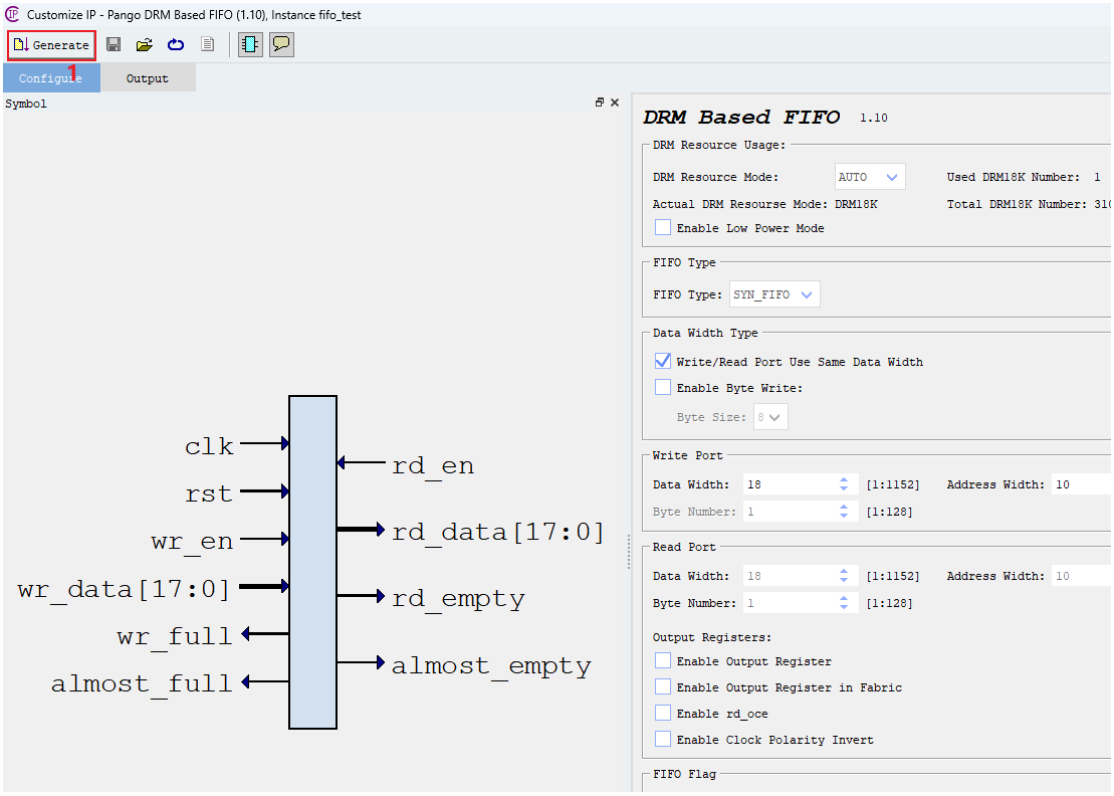


图 4.2-12

```
Create directory 'rtl' ...
Copy 'rtl\ipm2l_fifo_ctrl_v1_1.v.xml' to 'rtl' ...
Copy 'rtl\ipm2l_fifo_v1_10.v.xml' to 'rtl' ...
Copy 'rtl\ipm2l_sdpram_v1_10.v.xml' to 'rtl' ...
Compile file 'rtl\ipm2l_sdpram_v1_10.v.xml' to 'rtl\ipm2l_sdpram_v1_10_fifo_test.v' ...
Compile file 'rtl\ipm2l_fifo_v1_10.v.xml' to 'rtl\ipm2l_fifo_v1_10_fifo_test.v' ...
Compile file 'rtl\ipm2l_fifo_ctrl_v1_1.v.xml' to 'rtl\ipm2l_fifo_ctrl_v1_1_fifo_test.v' ...
Copy 'ipm2l_fifo_wrapper_v1_10.v.xml' ...
Compile file 'ipm2l_fifo_wrapper_v1_10.v.xml' to 'fifo_test.v' ...
Found top module 'fifo_test' in file 'fifo_test.v'.
Copy 'ipm2l_fifo_wrapper_v1_0_tb.v.xml' ...
Compile file 'ipm2l_fifo_wrapper_v1_0_tb.v.xml' to 'fifo_test_tb.v' ...
Run file 'creatResetValue.tcl' ...
Create template file 'fifo_test_tmpl.v' ...
Create template file 'fifo_test_tmpl.vhdl' ...
There are 4 source files to synthesize.
Synthesis is disabled.
Done: 0 error(s), 0 warning(s)
```

图 4.2-13

没有任何错误表示生成成功。

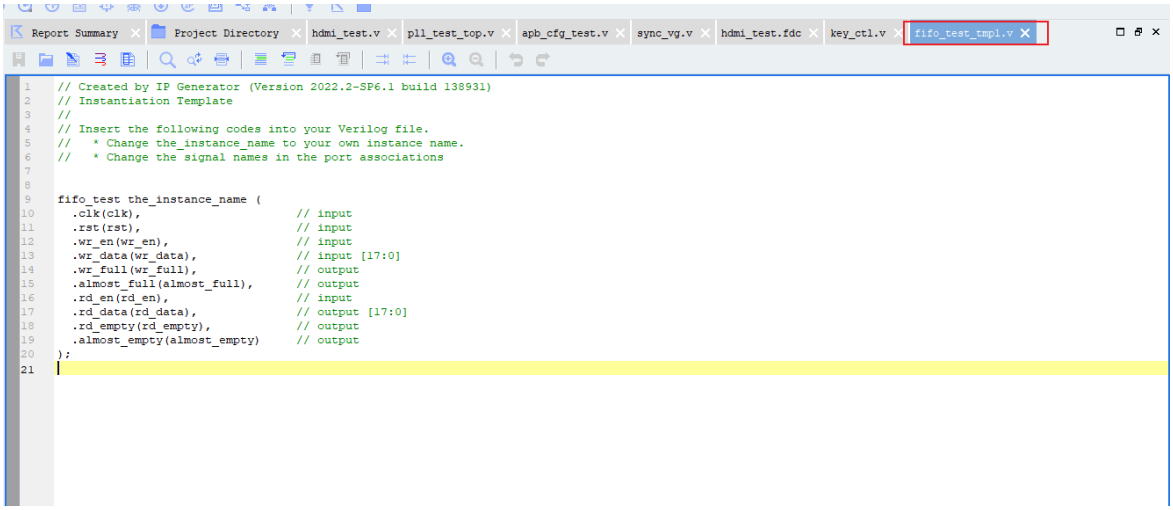


图 4.2-14

同时工具也会自动弹出一个IP的例化模板，供我们使用。只需要把该例化模板添加到自己的工程之中，即可使用我们生成的IP。

5.Pango 的时钟资源--锁相环

5. 1. 实验目的

了解Logos2系列的PLL的使用及配置方法。

5. 2. 实验原理

5.2.1.PLL 介绍

锁相环作为一种反馈控制电路，其特点是利用外部输入的参考信号来控制环路内部震荡信号的频率和相位。因为锁相环可以实现输出信号频率对输入信号频率的自动跟踪，所以锁相环通常用于闭环跟踪电路。锁相环在工作的过程中，当输出信号的频率与输入信号的频率相等时，输出电压与输入电压保持固定的相位差值，即输出电压与输入电压的相位被锁住，这就是锁相环名称的由来。

锁相环拥有强大的性能，可以对输入到FPGA的时钟信号进行任意分频、倍频、相位调整、占空比调整，从而输出一个期望时钟；除此之外，在一些复杂的工程中，哪怕我们不需要修改任何时钟参数，也常常会使用PLL来优化时钟抖动，以此得到一个更为稳定的时钟信号。正是因为PLL的这些性能都是我们在实际设计中所需要的，并且是通过编写代码无法实现的，所以PLL IP核才会成为程序设计中最常用IP核之一。

PLL IP是紫光同创基于PLL及时钟网络资源设计的IP，通过不同的参数配置，可实现时钟信号的调频、调相、同步、频率综合等功能。

5.2.2.IP 配置

首先点击快捷工具栏的“IP”图标，进入IP例化设置

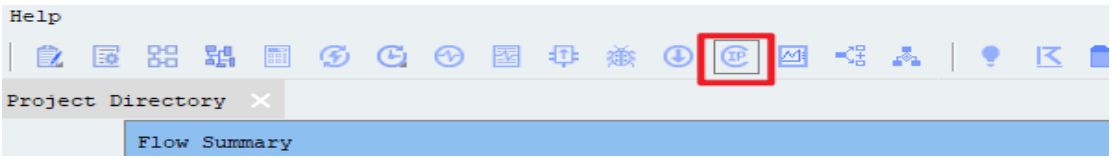


图 5.2-1 “IP”图标示意图

然后在IP目录处选择PLL，在Instance name处为本次实例化的IP取一个名字，接着点击Customise进入IP配置页面。操作示意图如下：

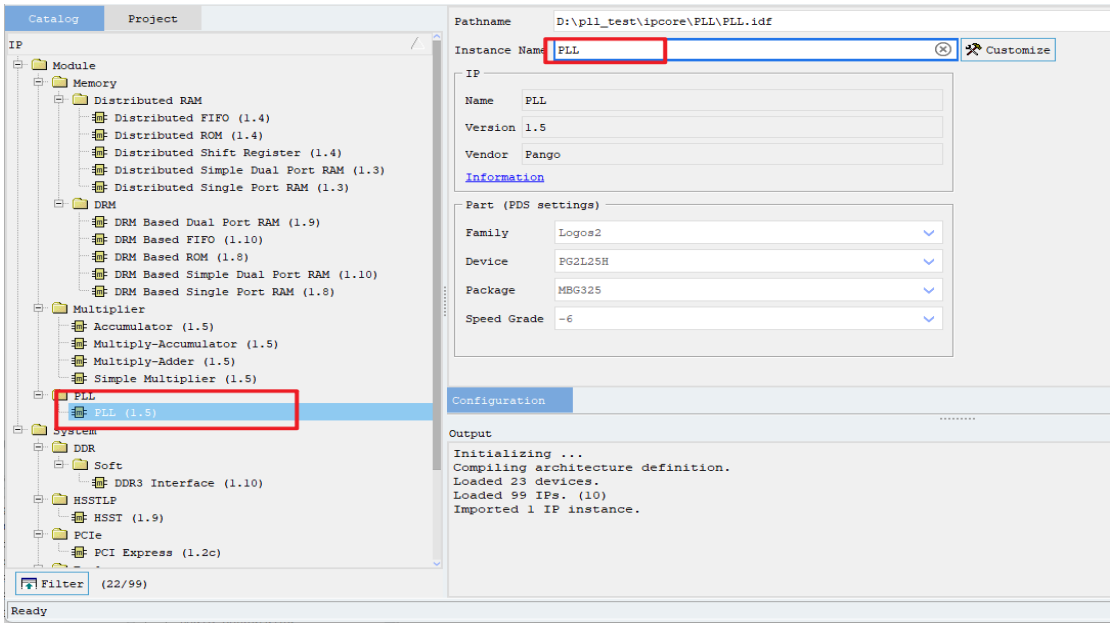


图 5.2-2 IP配置流程图1

PLL的使用可选择Basic和Advanced两种模式，Advanced模式下PLL的内部参数配置完全开放，需要自己填写输入分频系数、输出分频系数、占空比、相位、反馈分频系数等才能正确配置。Basic模式下用户无需关心PLL的内部参数配置，只需输入期望的频率值、相位值、占空比等，IP将自动计算，得到最佳的配置参数。如果没有特殊应用，建议使用Basic模式配置PLL。本次实验我们选择Basic Configuration。

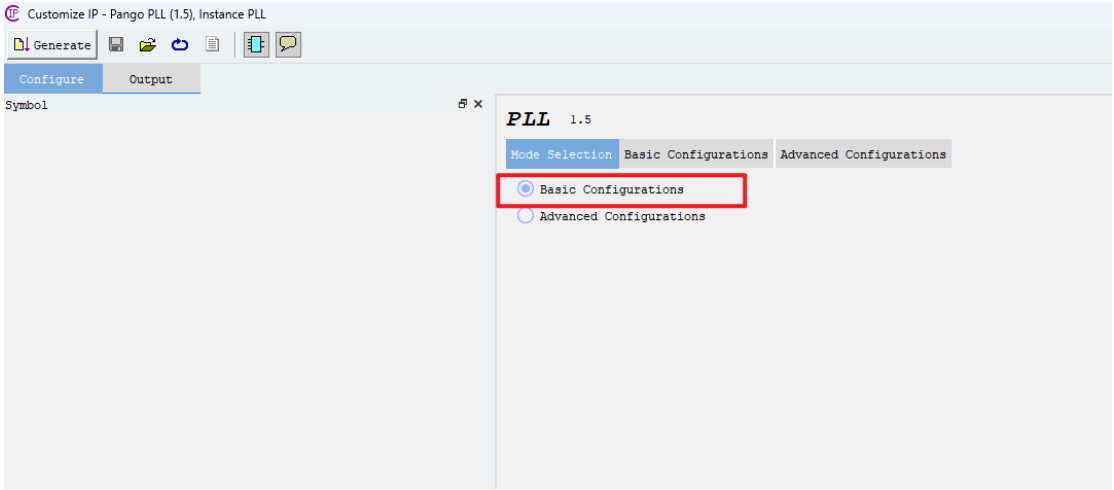


图 5.2-3 IP配置流程图2

接下来进行基础配置：

在Public Configurations一栏将输入时钟频率设置为27MHZ。

在Clockout0 Configurations选项卡下，勾选使能clkout0，将输出频率设置为54MHZ。

在Clockout1 Configurations选项卡下，勾选使能clkout1，将输出频率设置为81MHZ。

在Clockout2 Configurations选项卡下，勾选使能clkout2，将输出频率设置为81MHZ，并设置相位偏移为180度。

其他选项可以使用默认设置，若有其他需求可以查阅IP手册了解，本实验我们暂介绍IP基本的使用方法：

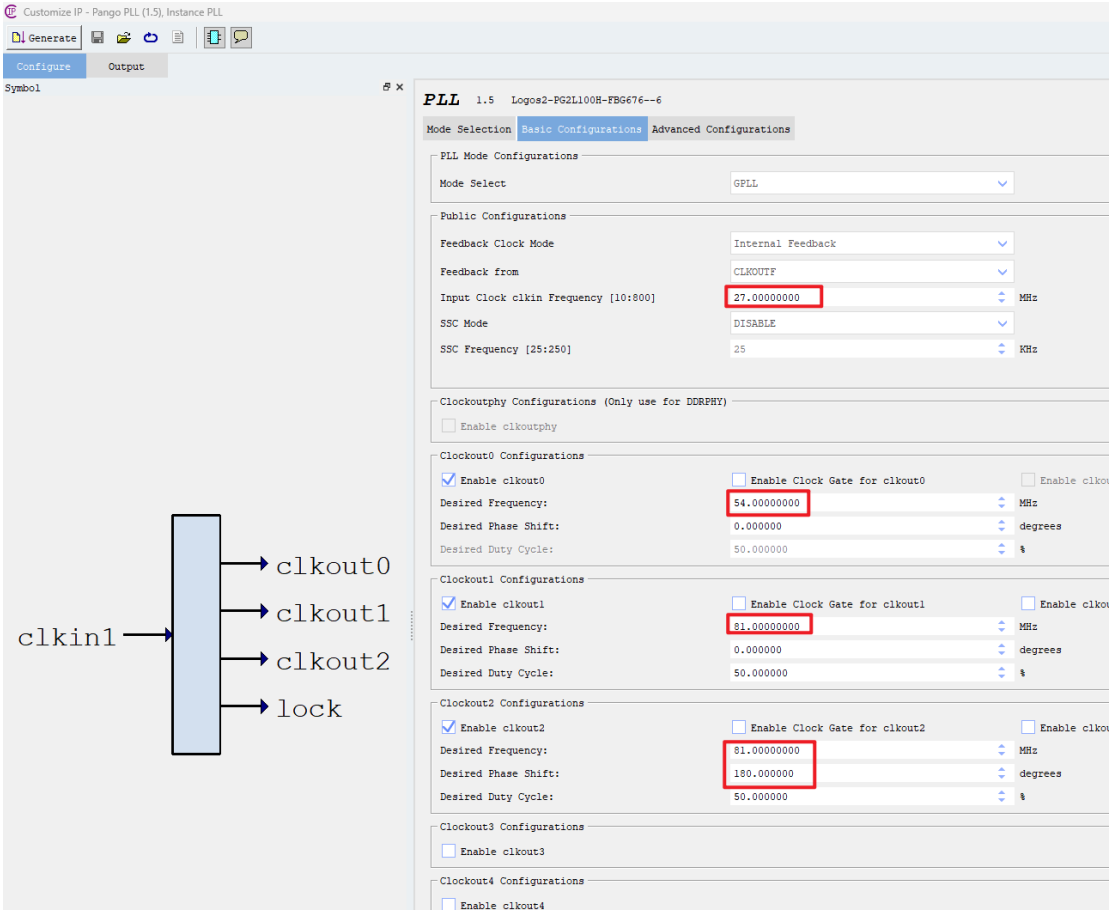


图 5.2-4 IP配置流程图3

点击左上角generate生成IP。

5.3. 代码设计

模块接口列表如下所示：

表 5.3-1 PLL IP使用实验模块接口表

端口	I/O	位宽	描述
sys_clk	input	1	系统时钟
clkout0	output	1	54MHZ时钟
clkout1	output	1	81MHZ时钟
clkout2	output	1	81MHZ时钟，相位偏移180度
lock	output	1	时钟锁定信号，当为高电平时，代表IP核输出时钟稳定。

PLL_TEST顶层代码：

```
1.  module PLL_TEST(
2.      input                   sys_clk         ,
3.      output                  clkout0         ,
4.      output                  clkout1         ,
5.      output                  clkout2         ,
6.      output                  lock
7.      );
8.
9.  PLL PLL_U0 (
10.     .clkout0                (clkout0        ),// output
11.     .clkout1                (clkout1        ),// output
12.     .clkout2                (clkout2        ),// output
13.     .lock                   (lock           ),// output
14.     .clkin1                 (sys_clk        ) // input
15. );
16.
17.
18. endmodule
```

该模块的功能是例化PLL IP核，功能简单，在此不做说明。

PLL_tb测试代码：

```
1.  timescale 1ns / 1ps
2.
3.  module      PLL_tb();
```

```
4.      reg                                     sys_clk                 ;
5.      wire                                    clkout0                 ;
6.      wire                                    clkout1                 ;
7.      wire                                    clkout2                 ;
8.      wire                                    lock                    ;
9.
10.
11.
12.     initial
13.         begin
14.             #2
15.                     sys_clk <= 0        ;
16.         end
17.
18.     parameter     CLK_FREQ = 27;//Mhz
19.     always # ( 1000/CLK_FREQ/2 ) sys_clk = ~sys_clk ;
20.
21.
22. PLL_TEST u_PLL_TEST(
23.     .sys_clk                            (sys_clk                ),
24.     .clkout0                            (clkout0                ),
25.     .clkout1                            (clkout1                ),
26.     .clkout2                            (clkout2                ),
27.     .lock                               (lock                   )
28. );
29.
30.
31. endmodule
```

timescale定义了模块仿真的时间单位和时间精度。时间单位是1纳秒，精度是1皮秒。

initial块负责初始化系统时钟。在仿真启动后的2纳秒，系统时钟sys_clk被设置为0。这是为了在仿真开始时定义一个已知的初始状态。

代码定义了一个时钟频率参数CLK_FREQ为27MHz，并使用一个always块来翻转系统时钟信号。always块中的逻辑使得sys_clk每37纳秒翻转一次，从而生成一个27MHz的方波时钟信号。这种时钟信号用于驱动被测试的PLL_TEST模块。

最后，将测试平台的各个信号连接到PLL_TEST模块。这包括将生成的系统时钟sys_clk连接到PLL_TEST的时钟输入端，并将PLL_TEST的输出信号clkout0、clkout1、clkout2和lock使用wire引出观察。

5.4. PDS 与 Modelsim 联合仿真

PDS支持与Modelsim或QuestaSim等第三方仿真器的联合仿真，而Modelsim是较为常用的仿真器，使用PDS与Modelsim来进行联合仿真。

接下来选择Project->Project Setting，打开工程设置，准备设置联合仿真。

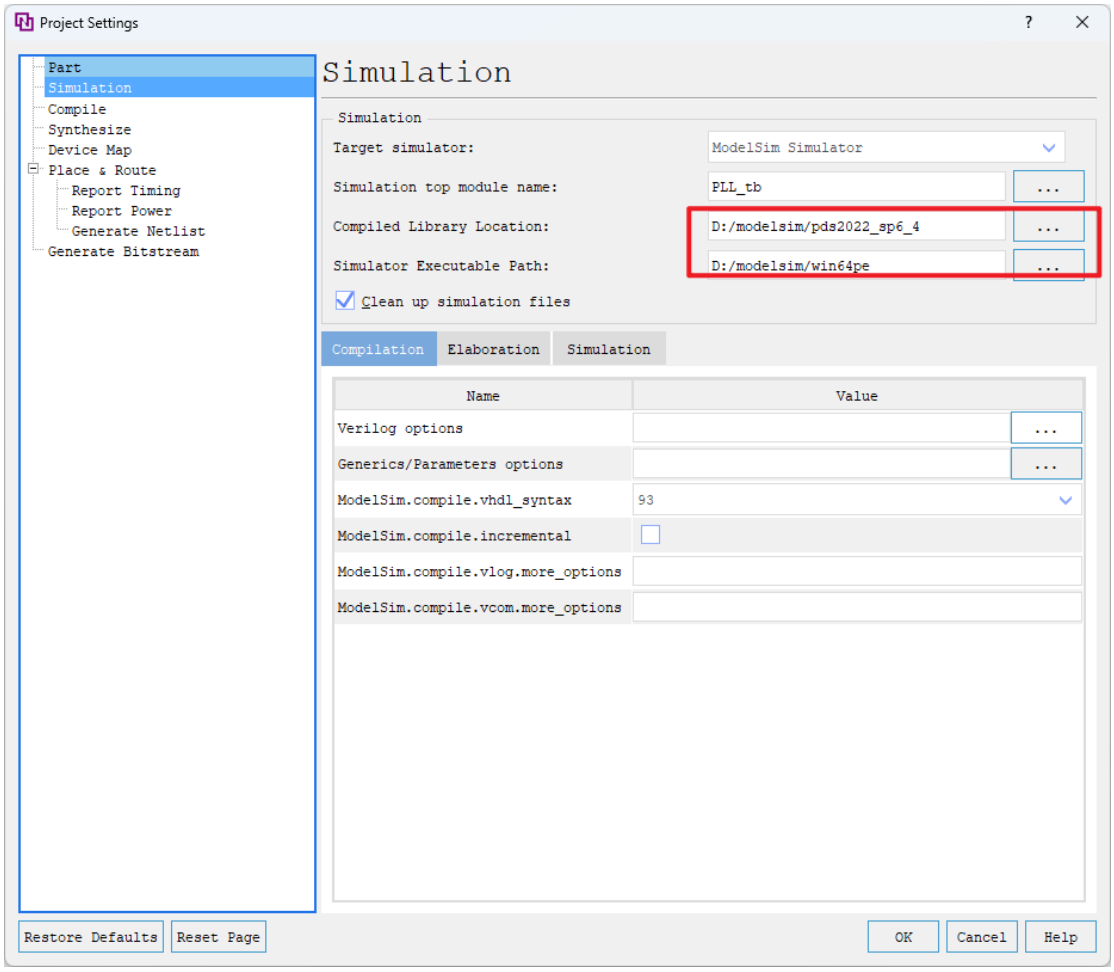


图 5.4-1 PDS和Modelsim联合仿真流程图4

选择Simulation选项卡,红框1选择刚才编译生成的仿真库的路径,红框2选择Modelsim的启动路径,之后点击OK。

右键仿真的文件,选择Run Behavior Simulation开始行为仿真。

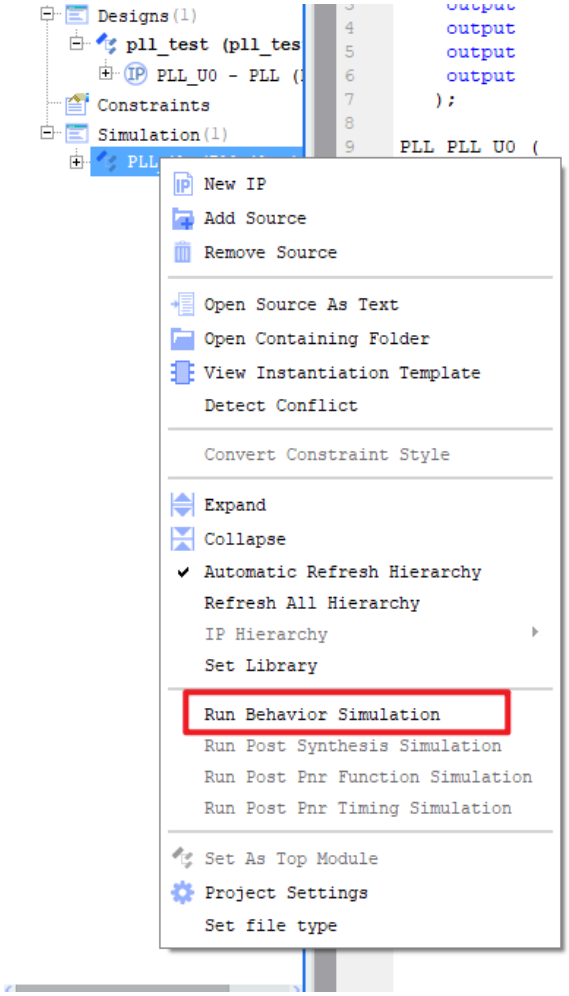


图 5.4-2 PDS和Modelsim联合仿真流程图5

运行后会自动打开Modelsim。并执行仿真,如果没有任何报错,则表示成功。如果出现错误，请检测PDS与Modelsim的配置。

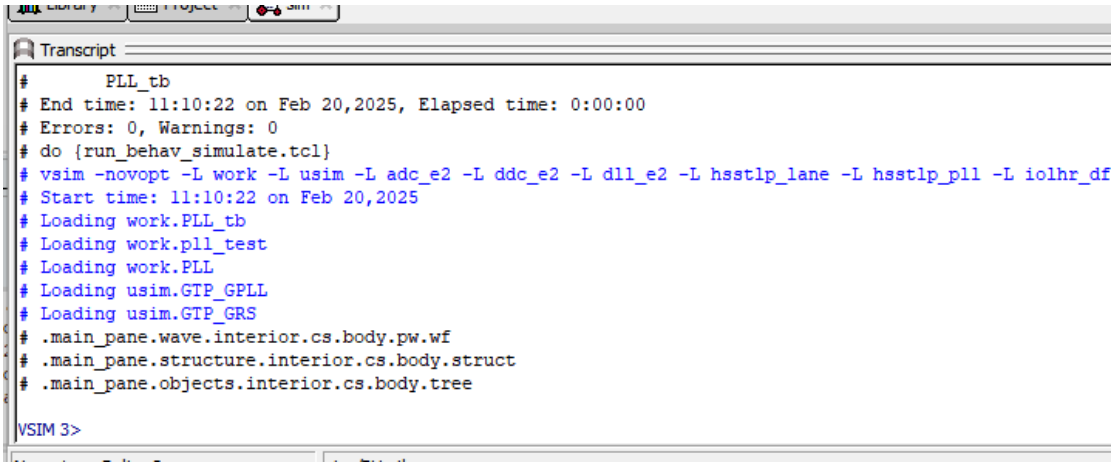


图 5.4-3 PDS和Modelsim联合仿真流程图6

5.5. 实验现象

点击Wave观察PLL输出信号：

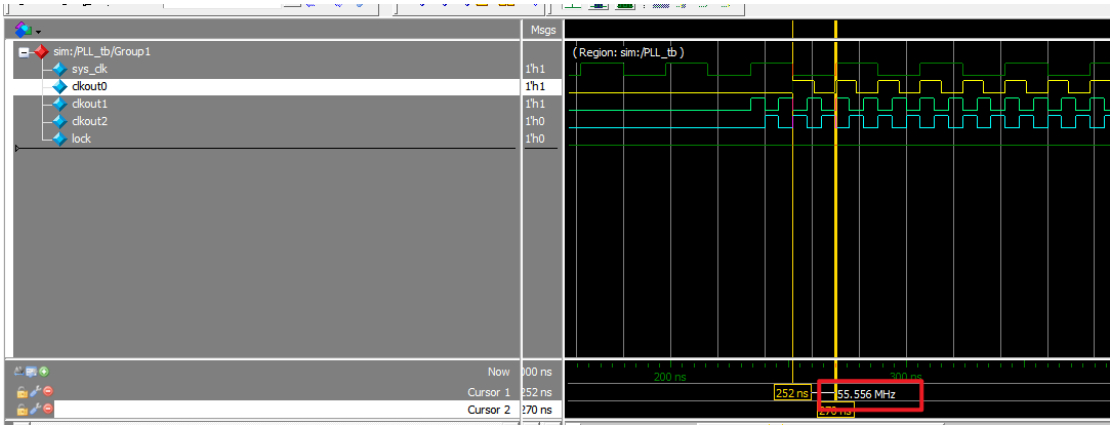


图 5.5-1 PLL IP使用实验结果波形图1

使用标尺测量clkout0，发现其一个时钟周期是18ns，也就是55.556MHZ。出现了偏差是因为tb生成的27MHZ其实并不准确，所以导致误差。

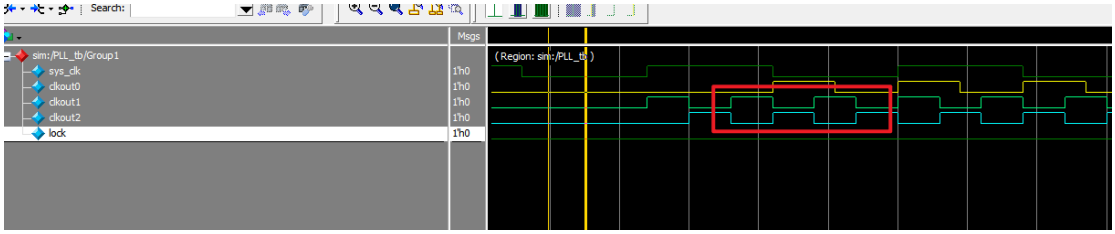


图 5.5-2 PLL IP使用实验结果波形图2

使用可以看到clkout1和clkout2相位偏差180°，符合设置。需要注意PLL的输出时钟应该在时钟锁定信号lock有效之后才能使用，lock信号拉高之前输出的时钟是不确定的。

6.Pango 的 ROM、RAM、FIFO 的使用

6. 1. 实验简介

实验目的：

掌握紫光平台的RAM、ROM、FIFO IP的使用

实验环境：

Window11

PDS2022.2

硬件环境：

暂无

6. 2. 实验原理

不管是Logos系列或者是Logos2系列，其IP配置以及模式和功能均一致，不会像PLL那样有动态配置以及内部反馈选项的选择等之间的差异，所以是RAM、ROM、FIFO是通用的。

6.2.1.RAM 介绍

RAM即随机存取存储器。它可以在运行过程中把数据写进任意地址，也可以把数据从任意地址中读出。其作用可以拿来做数据缓存，也可以跨时钟，也可以存放算法中间的运算结果等。

注意，PDS的IP配置工具中提供两种不同的RAM，一种是Distributed RAM(分布式RAM)另一种是DRM Based RAM，分布式RAM用的是LUT(查找表)资源去构成的RAM，这种RAM会消耗大量LUT资源，因此通常在一些比较小的存储才会用到这种RAM，以节省DRM资源。而DRM Based RAM是利用片内的DRM资源去构成的RAM，不占用逻辑资源，而且速度快，通常设计中均使用DRM Based RAM。

RAM分为三种，如下表所示：

表 6.2-1

RAM类型	特点
单端口RAM	只有一个端口可以读写。只有一个读写口和地址口
伪双端口RAM	有wr和rd两个端口，顾名思义，wr只能写，rd只能读

真双端口RAM	提供A和B两个端口，两个端口均可以独立进行读写

注意，当使用真双端口时，要避免出现同时读写同个地址，这会造成写入失败，在逻辑设计上需要避开这个情况。

以下给出比较常用的RAM的配置作为介绍，通常我们比较常用伪双端口RAM来设计，如图 6.2-1所示：

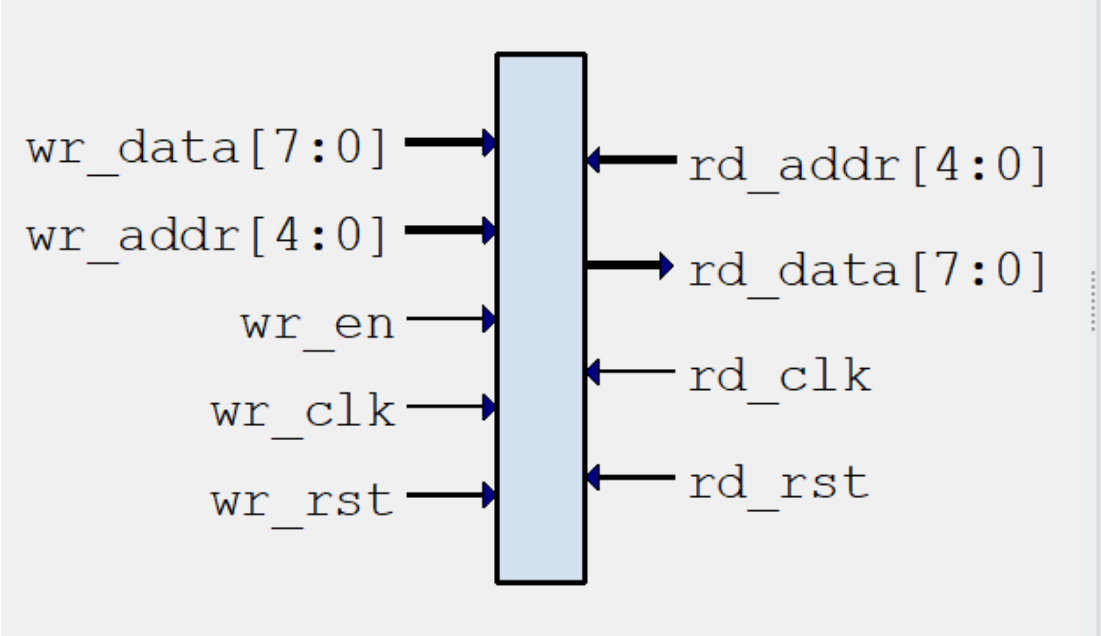


图 6.2-1

图 6.2-2为IP配置：

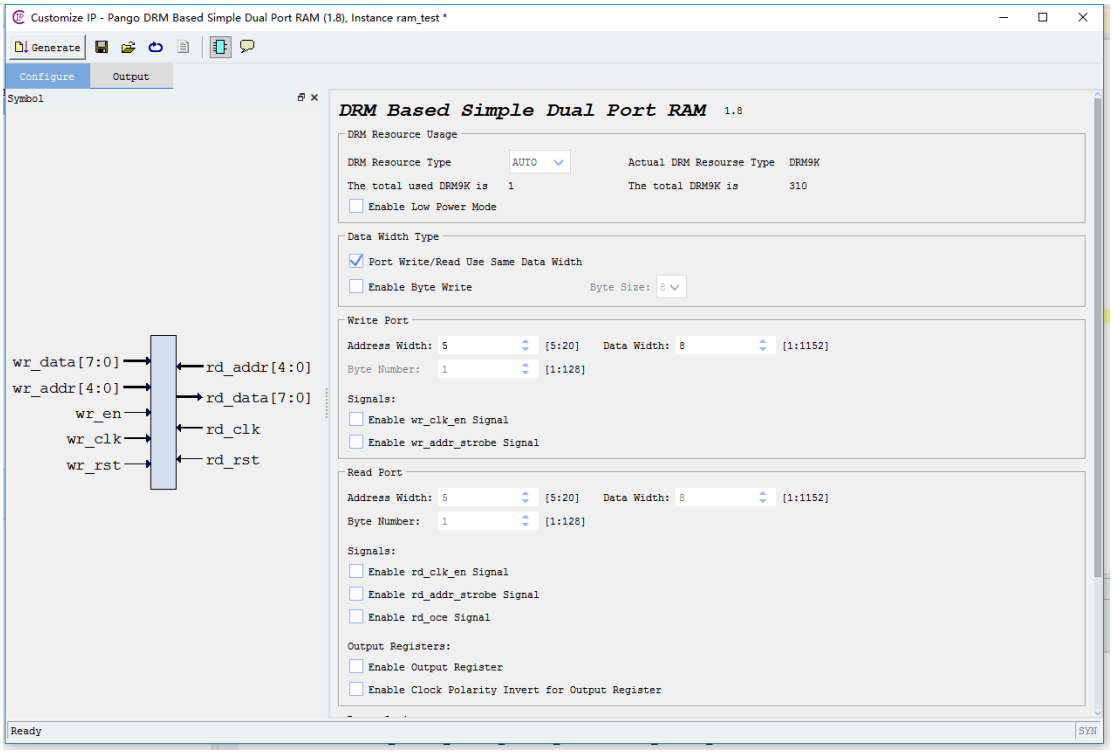


图 6.2-2

注意，如果勾选Enable Output Register(输出寄存)，输出数据会延迟一个时钟周期。

具体每个端口的含义这里参考官方手册，大家也可以自行查看IP手册，如下图所示:

端口名	输入/输出	说明
wr_data	输入	写数据信号，位宽范围1~1152
wr_addr	输入	写地址信号，位宽范围5~20
wr_en	输入	写使能信号 1：写使能 0：读使能
wr_clk	输入	写时钟信号
wr_clk_en	输入	写时钟使能信号 1：对应地址有效 0：对应地址无效
wr_rst	输入	写端口复位信号，高有效
wr_byte_en	输入	Byte Write使能信号，当配置“Enable Byte Write”选项勾选时有效，位宽范围1~128。 1：对应Byte值有效； 0：对应Byte值无效
wr_addr_strobe	输入	写地址锁存信号 1：对应地址无效，上一个地址被保持 0：对应地址有效
rd_data	输出	读数据信号，位宽范围1~1152
rd_addr	输入	读地址信号，位宽范围5~20
rd_clk	输入	读时钟信号
rd_clk_en	输入	读时钟使能信号。 1：对应地址有效； 0：对应地址无效。
rd_rst	输入	读端口复位信号，高有效
rd_oce	输入	读数据输出寄存使能信号 1：对应地址有效，读数据寄存输出 0：对应地址无效，读数据保持
rd_addr_strobe	输入	读地址锁存信号 1：对应地址无效，上一个地址被保持 0：对应地址有效

图 6.2-3

DRM Resource Type：用于配置所建 RAM IP核用的是哪种资源，不同芯片型号可选资源是不一样的，有的是9K,有的是18K,有的是36K,如果没有特殊情况，直接AUTO即可。

6.2.1.1. RAM 的读写时序

配置成不同模式的时候，RAM的读写时序是不一样的，真双端口和单端口的RAM配置均有三种模式，而伪双端口只有一种。由于真双端口和单端口的配置是一样的，这里以真双端口为例子。

分为NORMAL_WRITE(正常模式)、TRANSPARENT_WRITE(直写)、READ_BEFORE_WRITE(读优先模式)三种模式。

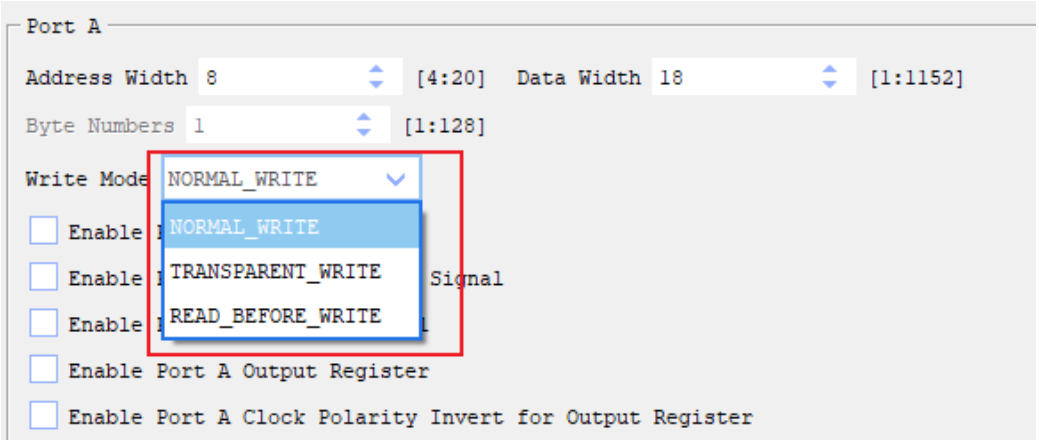


图 6.2-4

而伪双端口不属于上面三种模式，有它独特的模式。这几种模式的差异就在于读写时序的不同，接下来，我们来分析读写时序。

以下时序图均来自官方IP手册，并且均未使能输出寄存。注意wr_en为1时表示写数据，为0表示读数据。

6.2.1.1.1.NORMAL_WRITE

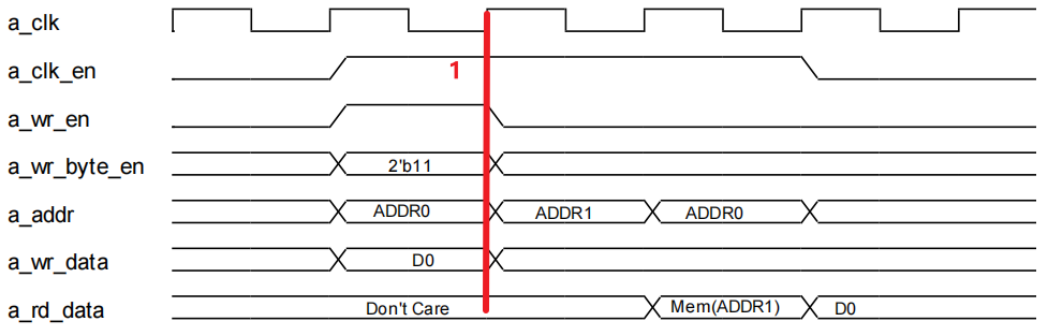


图 6.2-5

在NORMAL_WRITE这种模式下，可以看到，当时钟的上升沿到来，且clk_en和wr_en均为高电平时，就会把数据写到对应的地址里面，如图中的1时刻。然后看读数据端口，当wr_en不为0的时候，a_rd_data一直为Don't Care状态，而当时钟上升沿到来，且clk_en为高电平，wr_en为低电平时，a_rd_data输出当前a_addr里的数据，即Mem(ADDR1)和ADDR0里的D0。

6.2.1.1.2.READ_BEFORE_WRITE

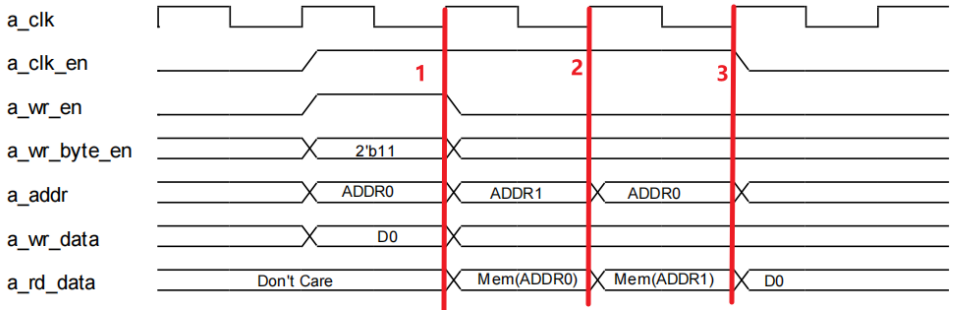


图 6.2-6

在READ_BEFORE_WRITE这种模式下，可以看到在1的时刻，时钟上升沿到来，且clk_en和wr_en均为高电平，D0写进了ADDR0里面，但是注意看此时的a_rd_data和a_addr，可以发现，此时a_wr_en并不为0，可a_rd_data还是输出了上一刻ADDR0的数据(因为不是输出D0)。之后，a_wr_en拉低，此时才是读数据，在3时刻，把ADDR0的数据读出来，a_rd_data才输出了D0。

所以总结一下，这个模式其实就是进行写操作时，读端口会把当前写的地址的原始数据输出，因此叫读优先模式很合情合理对吧，顾名思义，就是我优先把原来的数据读出来。

6.2.1.1.3.Transparent_Write

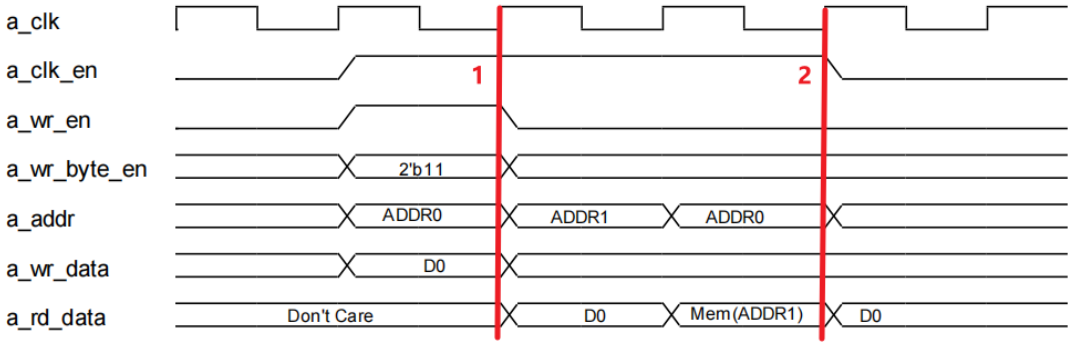


图 6.2-7

在Transparent_Write这种模式下，可以看到在1的时刻，时钟上升沿到来，且 clk_en 和wr_en均为高电平，D0写进了ADDR0里面，但是注意看此时的a_rd_data和a_addr，可以发现，此时a_wr_en并不为0，可a_rd_data居然直接输出了D0，之后a_wr_en拉低，进入读状态，在2时刻，再一次把ADDR0的数据读出来，输出了D0。

分析总结一下，根据1时刻的情况，我们可以得出结论，在这种模式下，当我们进行写操作时，读端口会马上输出我们写入的数据。所以叫直写模式。

6.2.1.1.4.伪双端口的读写时序

注意：wr_en为1时是写操作，为0是读操作。

伪双端口的读写时序与上面三种都不同，我们看图8的时序来分析：

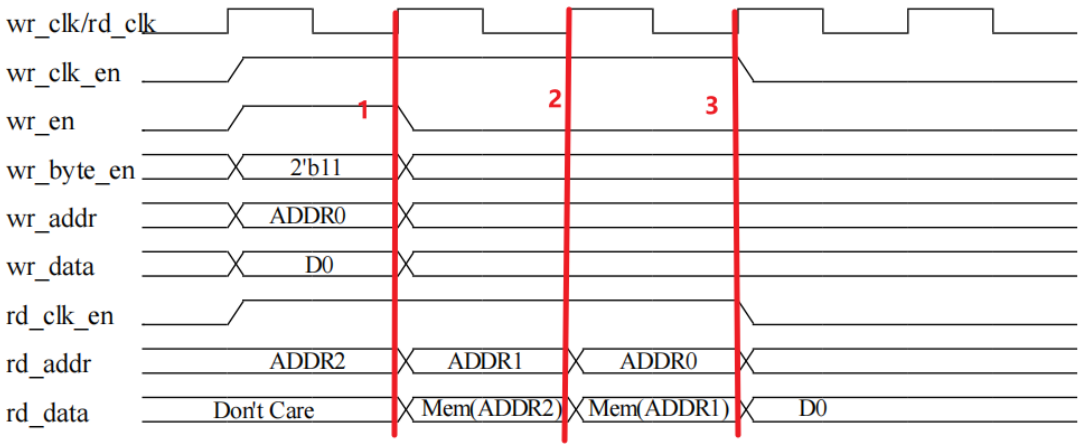


图 6.2-8

注意看1时刻，此时wr_en和wr_clk_en均为高电平，所以是写操作，所以1时刻就是往地址ADDR0里写入D0，注意此时的rd_addr和rd_data，可以看到这一时刻rd_addr是ADDR2，然后进行写操作时，rd_data同样输出了ADDR2里的数据，而此时wr_en还是高

电平。接下来看2和3时刻，此时wr_en为0，rd_clk_en是高电平，所以是读操作，此时分别读出ADDR1和ADDR0里的数据，之后rd_clk_en变成低电平，读时钟无效，可以看到rd_data保持D0输出。

分析总结一下，主要是1时刻，大家可以看到1时刻往ADDR0写入了D0，读端口却输出了ADDR2中的数据。仔细观察可以得出结论：伪双端口RAM在进行写操作的时候，会把当前读端口指向的地址的数据输出。是不是有点像直写？只不过直写是输出写入的数据，而伪双端口是输出读端口指向的地址的数据。

具体大家可以结合视频讲解。

2.1.1.2 ROM介绍

ROM即只读存储器，在程序的运行过程中他只能被读取，无法被写入，因此我们应该在初始化的时候就给他配置初值，一般是在生成IP的时候通过导入.dat文件对其进行初值配置。

注意，PDS的IP配置工具中提供两种不同的ROM，一种是Distributed ROM(分布式ROM)另一种是DRM Based ROM，分布式ROM用的是LUT(查找表)资源去构成的ROM，这种ROM会消耗大量LUT资源，因此通常在一些比较小的存储才会用到这种RAM，以节省DRM资源。而DRM Based ROM是利用片内的DRM资源去构成的ROM，不占用逻辑资源，而且速度快，通常设计中均使用DRM Based ROM。

以下给出比较常用的ROM的配置作为介绍，由于只能读，因此其均为单端口ROM如图6.2-9所示：

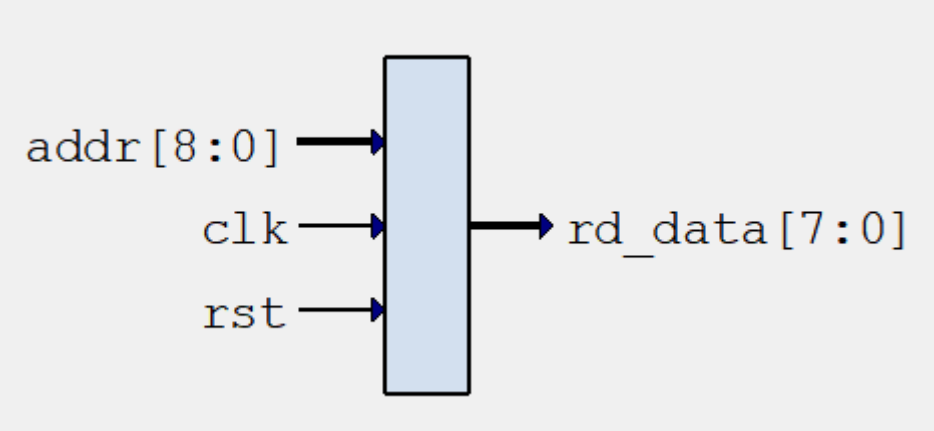


图 6.2-9

下图为IP配置：

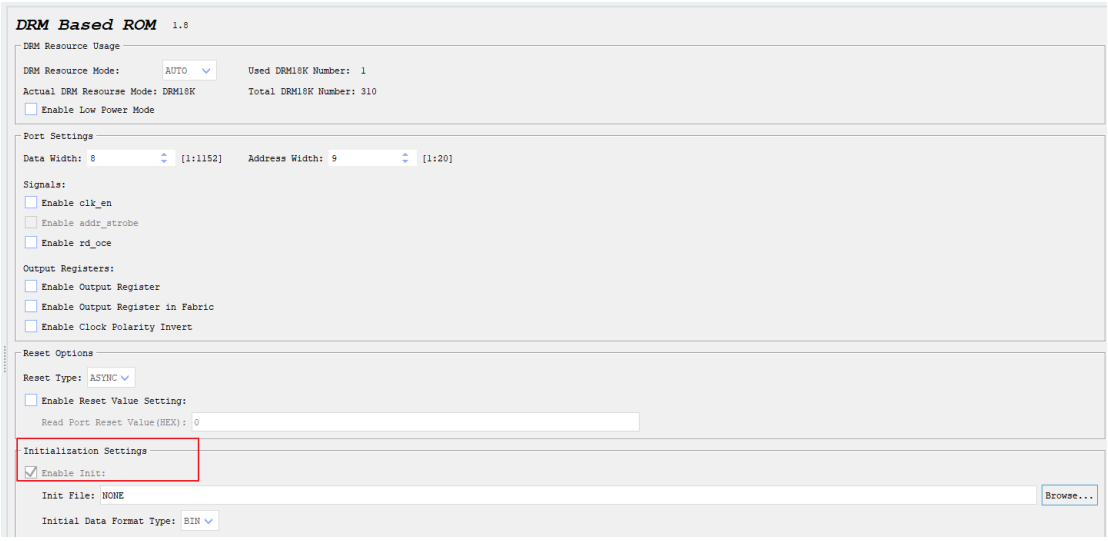


图 6.2-10

注意，如果勾选Enable Output Register(输出寄存)，输出数据会延迟一个时钟周期。

同时，可以看到Enable Init选项是默认勾选的，并且不可取消。

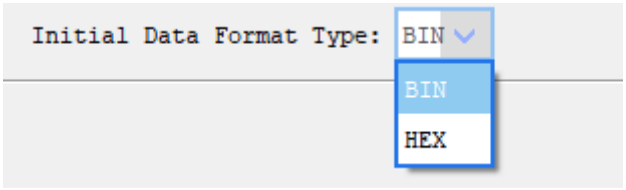


图 6.2-11

导入的数据的格式只能为二进制或者是十六进制。

具体每个端口的含义这里参考官方手册，大家也可以自行查看IP手册，如图 6.2-12所示:

端口	I/O	描述
addr	I	读地址信号。
addr_strobe	I	读地址锁存信号。 1：对应地址无效，地址被保持； 0：对应地址有效。
rd_data	O	读数据信号。
clk	I	时钟信号。
clk_en	I	时钟使能信号。 1：对应地址有效； 0：对应地址无效。
rst	I	复位信号。 1：复位； 0：复位释放。
rd_oce	I	输出寄存使能信号。 1：读数据寄存输出； 0：寄存输出数据保持。

图 6.2-12

可以看到图 6.2-12给出的是完整的接口列表，一般我们只需要addr、rd_data、clk、rst这四个信号即可。

以下时序图均来自官方IP手册，并且均未使能输出寄存。

6.2.1.2.ROM 的读时序

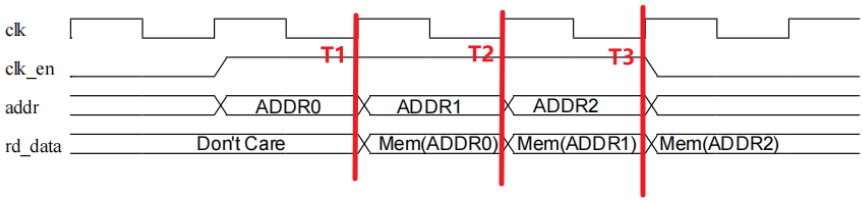


图 6.2-13

可以看到该时序是非常简单的，比如在TI时刻，当clk上升沿到来时，且clk_en为高电平时，给出要读出的地址，rd_data就会输出数据，在不勾选输出使能寄存的情况下，rd_data的输出会有延迟，具体时间可以从仿真里看到，所以我们在下个时钟周期的上升沿即T2时刻的上升沿才能获取到ROM读出的值。

所以整体时序非常简单，如果勾选了clk_en信号，就要给clke_en高电平才能读数据，如果不勾选clk_en信号，就一直根据地址读取ROM数据。

6.2.2.FIFO 介绍

FIFO即先入先出，在FPGA中，FIFO的作用就是对存储进来的数据具有一个先入先出特性的一个缓存器，经常用作数据缓存或者进行数据跨时钟域传输。FIFO和RAM最大的区别就是FIFO不需要地址，采用的是顺序写入，顺序读出。

在紫光的IP工具中又分为Distribute FIFO和DRM FIFO，其实就是用不同的资源去构成，前者Distribute FIFO也就是分布式FIFO，使用的是片上的LUT资源去构成，而DRM FIFO使用的是片上的DRM资源去构成，DRM构成的FIFO其性能大于LUT资源构成的，不仅容量更大，且可配置更多功能。

本章着重介绍DRM Based FIFO。

注意：FIFO 写满后禁止继续写入数据，否则将会写溢出。

注意：FIFO 读空后禁止继续读数据，否则将会读溢出。

以下给出常用的FIFO的配置作为介绍。

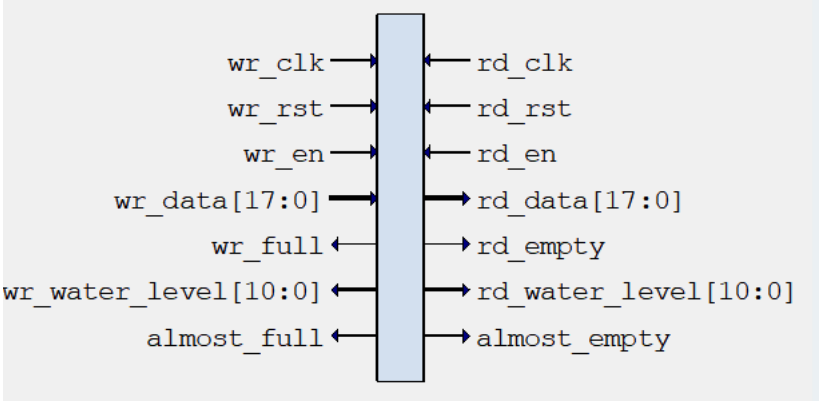


图 6.2-14

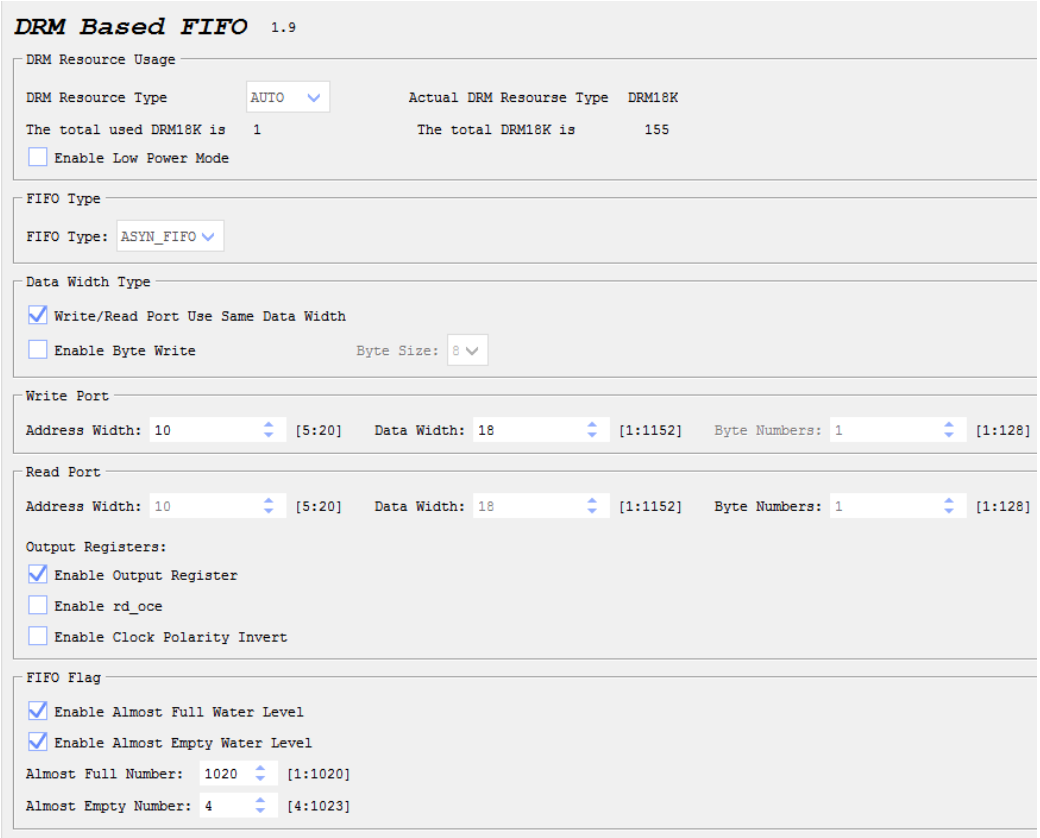


图 6.2-15

注意，如果勾选Enable Output Register(输出寄存)，输出数据会延迟一个时钟周期。

FIFO Type有SYNC和ASYNC两种，第一种是同步FIFO，读写端口共用一个时钟和复位，另一种是异步FIFO，读写时钟和复位均独立。在平常设计中，比较常用的是异步FIFO，因为同步FIFO和异步FIFO的读写时序一模一样，只有读写端口的时钟复位有差异，当异步FIFO的读写端口使用相同的时钟和复位，此时异步FIFO和同步FIFO基本是一致的。

Reset Type也可以选择SYNC和ASYNC两种，SYNC模式下需要时钟的上升沿采样到复位有效才会复位，而在ASYNC模式下，复位一旦有，FIFO立即复位。

其余端口说明引用官方IP手册，如图 6.2-16所示：

端口名	输入/输出	说明
wr_data	输入	写数据信号，位宽范围1~1152
wr_en	输入	写使能信号，高有效
wr_byte_en	输入	Byte Write使能信号，当配置“Enable Byte Write”选项勾选时有效，位宽范围1~128。 1：对应Byte值有效； 0：对应Byte值无效
clk	输入	同步FIFO时钟信号，仅同步FIFO有效
rst	输入	同步FIFO复位信号，高有效，仅同步FIFO有效
wr_clk	输入	异步FIFO写时钟信号，仅异步FIFO有效
wr_rst	输入	异步FIFO写复位信号，高有效，仅异步FIFO有效
wr_full	输入	FIFO Full信号 1：FIFO满 0：FIFO未满
almost_full	输出	FIFO Almost Full信号 1：FIFO将满 0：FIFO未将满
wr_water_level	输出	写端口water level信号，位宽范围5~20，表示写数据水位
rd_data	输出	读数据信号
rd_en	输出	读使能信号
rd_clk	输入	异步FIFO读时钟信号，仅异步FIFO有效
rd_rst	输入	异步FIFO读复位信号，仅异步FIFO有效
rd_empty	输入	FIFO Empty信号 1：FIFO空 0：FIFO未空
almost_empty	输出	FIFO Almost Empty信号 1：FIFO将空 0：FIFO未将空
rd_water_level	输出	读端口water level信号，位宽范围5~20，表示读数据水位
rd_oce	输入	输出寄存使能信号 1：对应地址有效，读数据寄存输出 0：对应地址无效，读数据保持

图 6.2-16

其中rd_water_level和wr_water_level分别代表”可读的数据量”和”已写入的数据量”，其含义与Xilinx的FIFO的wr_data_count和rd_data_count是一致的。

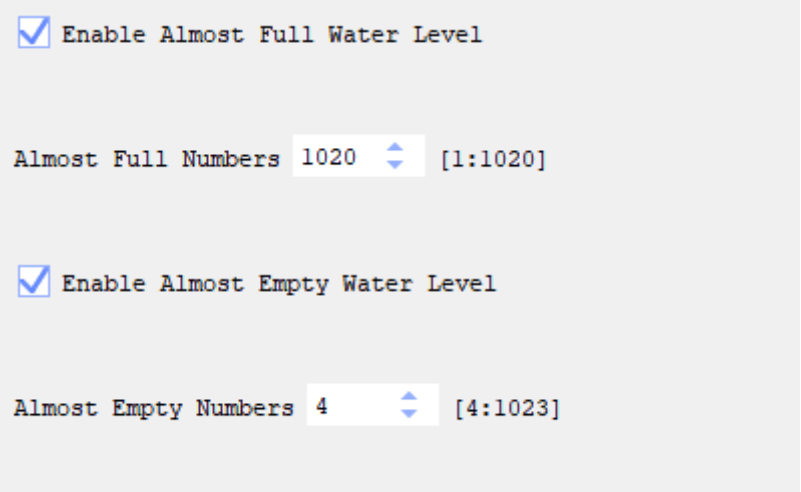


图 6.2-17

当我们将Enable Almost Full Water Level和Enable Almost Empty Water Level勾选上，才能看到rd_water_level和wr_water_level，而下面的Almost Full Numbers的设置是表示当写入1020个数据时，Almost Full信号就会拉高，Almost Empty Numbers的设置表示当可读数据剩下4个时Almost Empty信号就会拉高。

同时FIFO支持混合位宽，例如写端口16bit，读端口8bit。如果写入16'h0102，那么读出来会是8'h02,8'h01，会先读出低位。

如果写端口8bit，读端口16bit。当写入8'h01,8'h02时，读出来是16'h0201，先写入的数据存放在低位。

6.2.2.1.的读写时序

因为同步FIFO和异步FIFO的读写时序一致，这里用异步FIFO的读写时序图来做介绍。

注意：复位时高电平有效。读出数据均未勾选Enable Output Register(输出寄存)。

6.2.2.1.1.FIFO 未满时的写时序

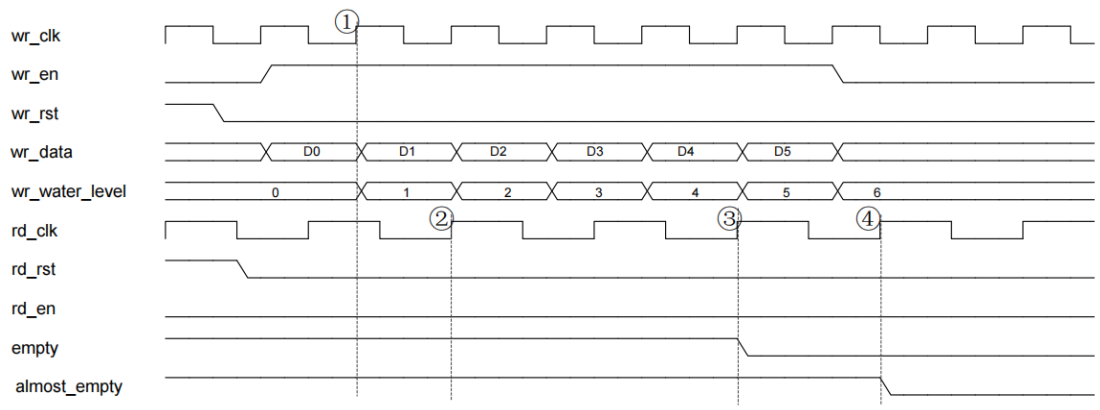


图 6.2-18

可以看到在1时刻，复位信号时低电平，处于工作状态，此时在wr_clk的上升沿且wr_en为高电平时将数据D0写入FIFO，wr_water_level也从0变1，表示已经写入了一个数据，此时注意看读端口的empty信号，在3时刻empty信号从高变低，意味着读端口已经有数据可以读了，FIFO不再为空，而注意看，rd_clk和wr_clk是不一样的，从1写入到3时刻empty拉低时，经过了3个rd_clk。

所以这里我们可以得出结论:rd_water_level要滞后wr_water_level三个rd_clk。

6.2.2.1.2.FIFO 将满时的写时序

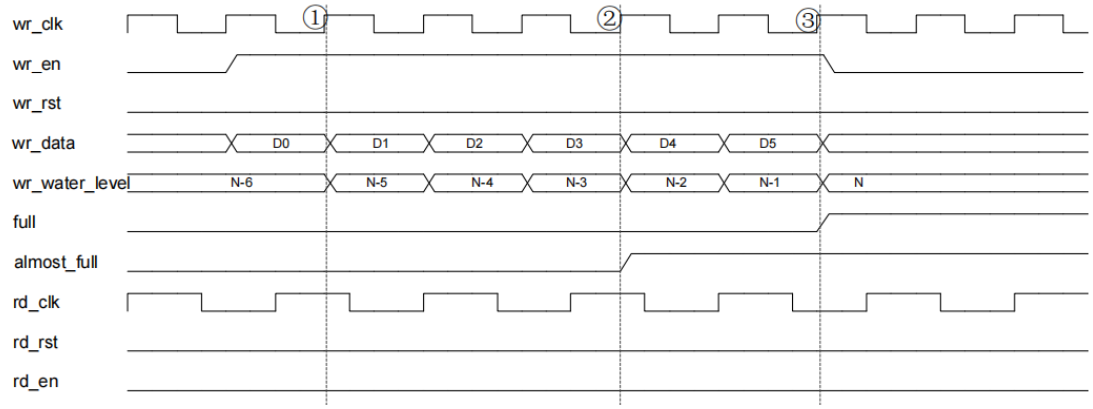


图 6.2-19

将满时主要分析full和almost_full信号。假设Almost Full Numbers设置为N-2，在1时刻，此时已经写入了N-6个数据，意味着再写6个数据FIFO就满了，从1时刻到2时刻一共写入了4个数据，因此当wr_water_level变成N-2时，满足条件，可以看到Almost Full信号拉高，再写两个数据FIFO就满了，所以再经过两个时钟周期后，Full信号拉高。

6.2.2.1.3.FIFO 在满状态下的读时序

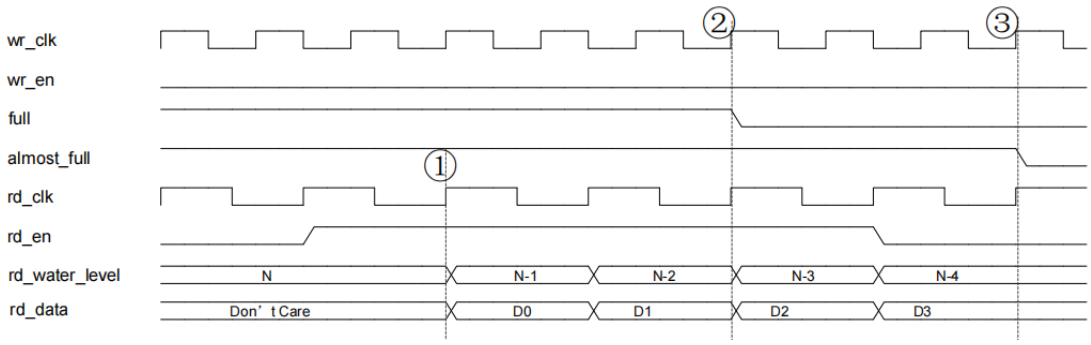


图 6.2-20

在满状态下，FIFO已经有N个数据了，此时在1状态下，rd_clk的上升沿，且rd_en为高电平时，此时从FIFO里读出数据(数据的输出有延时，仿真中延时0.2ns)。此时rd_water_level变成N-1，rd_data输出D0。然后看2时刻，full信号拉低，此时可以看以下，在1时刻到2时刻期间一共经过了3个wr_clk写端口才能判断到此时数据量已经不为满。所以我们可以得出结论，wr_water_level要滞后rd_water_level三个wr_clk。

6.2.2.1.4.FIFO 将空时的读时序

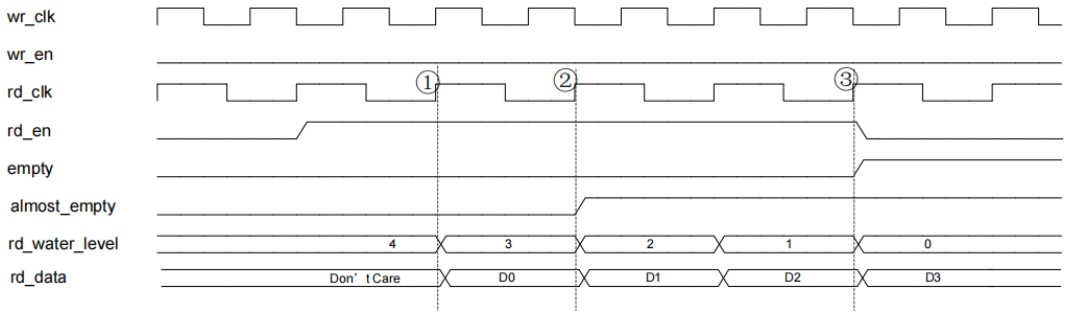


图 6.2-21

在1时刻，可读的数据量剩下4，假设Almost Empty Number设为2，在1时刻和2时刻分别读出了两个数据，所以在2时刻下，可读数据量剩下两个，达到Almost Empty Number触发条件，因此almost_empty信号拉高，再过两个时钟周期，即再读两个数据，FIFO将变成空状态，也就是状态3，此时empty信号拉高。

6. 3. 接口列表

该部分介绍每个顶层模块的接口。

ram_test_top.v

端口	I/O	位宽	描述
wr_clk	input	1	写时钟
rd_clk	input	1	读时钟
rst_n	input	1	全局复位

rw_en	input	1	1:写操作 0:读操作
wr_addr	input	5	写地址
rd_addr	input	5	读地址
Wr_data	input	8	写入RAM的数据
Rd_data	output	8	从RAM读出的数据

rom_test_top.v

端口	I/O	位宽	描述
rd_clk	input	1	读时钟
rst_n	input	1	全局复位
rd_addr	input	10	读地址
rd_data	input	64	从ROM读出的数据

fifo_test_top.v

端口	I/O	位宽	描述
sys_clk	input	1	写/读时钟
rst_n	input	1	全局复位
wr_data	input	8	写入FIFO的数据
wr_en	input	1	写使能
rd_en	input	1	读使能

wr_water_level	output	8	已写入FIFO的数据量
rd_water_level	output	8	可从FIFO读出的数据量
Rd_data	output	8	从FIFO读出的数据

6.4. 工程说明

暂无

6.5. 代码仿真说明

本次的顶层模块实际就是例化IP，然后把端口引出而已，主要代码都在testbench里面，所以我们直接介绍仿真代码。

6.5.1. RAM 仿真测试

```
1.  `timescale 1ns/1ns
2.  module ram_test_tb();
3.  reg     sys_clk;
4.  reg     rd_clk ;
5.  reg     rst_n;
6.  reg     rw_en;      //读写使能信号
7.
8.  reg     [7:0]     wr_data;
9.  reg     [4:0]     wr_addr;
10. reg     [4:0]     rd_addr;
11.
12. wire    [7:0]     rd_data;
13.
14. reg     [1:0]     state;
15.
16. initial
17. begin
18.     rst_n     <=     1'd0;
19.     sys_clk   <=     1'd0;
20.     rd_clk    <=     1'd0;
21.     #20
22.     rst_n     <=     1'd1;
23.
24. end
25.
26. //读写控制
27. always@(posedge sys_clk or negedge rst_n)     begin
28.     if(!rst_n)
29.     begin
30.         state     <=     2'd0;
31.         wr_data   <=     8'd0;
32.         rw_en     <=     1'd0;
```

```
            wr_addr   <=      8'd0;
            rd_addr   <=      8'd0;
        end
        else
        begin
        case(state)
            2'd0:begin
                    rw_en      <=      1'd1;
                    state      <=      2'd1;
                end

            2'd1:begin
                    if(wr_addr == 5'd31)
                    begin
                        rw_en      <=      1'd0;
                        state      <=      2'd2;
                        wr_data    <=      8'd0;
                        wr_addr    <=      5'd0;
                        rd_addr    <=      5'd0;
                    end
                    else
                    begin
                        state      <=      2'd1;
                        wr_data    <=      wr_data+1'b1;
                        rd_addr    <=      rd_addr+1'b1;
                        wr_addr    <=      wr_addr+1'b1;
                    end
                end
            2'd2:begin
                    if(rd_addr == 5'd31)
                    begin
                        state      <=      2'd3;
                        rd_addr    <=      5'd0;
                    end
                    else
                    begin
                        state      <=      2'd2;
                        rd_addr    <=      rd_addr+1'b1;
                    end
                end
            2'd3:begin
                    state      <=          2'd0;
                end

          default:      state      <=      2'd0;
          endcase
        end
end

//50MHZ
always#10 sys_clk = ~sys_clk;

//
GTP_GRS GRS_INST(
      .GRS_N(1'b1)
```

```
88.         ) ;
89.
90.   ram_test_top u_ram_test_top(
91.         .wr_clk     ( sys_clk     ),
92.         .rd_clk     ( sys_clk     ),
93.         .rst_n      ( rst_n       ),
94.         .rw_en      ( rw_en       ),
95.         .wr_addr    ( wr_addr     ),
96.         .rd_addr    ( rd_addr     ),
97.         .wr_data    ( wr_data     ),
98.         .rd_data    ( rd_data     )
99.   );
100.  endmodule
```

涉及到testbench的一些基础操作这里就不再详细讲解，只关注重点逻辑部分。从代码的**27行到80行**是RAM的读写控制状态机，主要用来控制读写地址的生成、读写使能信号以及写入数据的过程。

首先看**38-42行**，也就是**state=0**的时候，这里把rw_en拉高，并跳转到状态1，表示进入写操作阶段（这里没有时钟使能clk_en，可以不用管）。注意这是时序逻辑，信号的赋值要等到下一个时钟周期才生效，所以虽然在state=0时已经拉高了rw_en，但真正的数据写入是从下一个时钟周期才开始的。

接下来是**state=1**的逻辑，对应**44-60行**。在这个状态下，仿真一直在往RAM里面写数据。可以看到，当wr_addr不等于31时，每个周期wr_data和wr_addr都会不断加1（rd_addr也跟着加1，这是为了配合验证伪双口RAM的时序）。当wr_addr等于31的时候，会在下一个时钟周期把数据清零、地址清零并切换到state=2。但要注意，在当前时钟周期下还是会再往地址31写一次数据。这样整个写过程一共写入了32个数据，从地址0写到地址31。

然后进入**state=2**，对应**61-72行**，此时开始读数据。在这个状态下，每个周期上升沿rd_addr不断累加，直到等于31的时候，在下一个时钟周期才会清零并切换状态，而当前时钟周期还会继续把地址31的数据读出来。这样就保证了完整地读出地址0~31的32个数据。

最后是**state=3**，对应**73-75行**。这里可以看到只是等待一个时钟周期，然后再跳转回state=0，起到一个延时的作用，相当于让读写过程周而复始地循环执行。

整体来说，这个状态机的功能就是：**先写满32个数据到RAM，再依次读出32个数据，然后等待一个周期后重新开始**。如果是初学者，可能会对“在 rd_addr=31时还会再读一个数据”感到困惑，这其实就是时序逻辑的特性——在时钟上升沿触发的赋值，要等到下一个时钟周期才会生效，所以当rd_addr=31时，当前周期的读地址还是有效的，因此会再读出最后一个数据。

一句话总结：**时序逻辑的赋值总在下一个时钟周期才生效，所以写操作和读操作在边界条件时（wr_addr=31、rd_addr=31），依然会在当下周期完成最后一个数据的写入或读出。**

6.5.2.ROM 仿真测试

```
1.  `timescale 1ns/1ns
2.  module rom_test_tb();
3.  reg       sys_clk;
4.  reg       rst_n;
5.  reg       [9:0]       rd_addr;
6.  wire      [63:0]       rd_data;
7.
8.  initial
9.  begin
10.       rst_n       <=       1'd0;
11.       sys_clk     <=       1'd0;
12.       #20
13.       rst_n       <=       1'd1;
14.
15. end
16.
17. //50MHZ
18. always#10 sys_clk = ~sys_clk;
19. //
20. GTP_GRS GRS_INST(
21.       .GRS_N(1'b1)
22.       ) ;
23.
24. always@(posedge sys_clk or negedge rst_n)       begin
25.       if(!rst_n)
26.             rd_addr       <=       10'd0;
27.       else
28.             rd_addr       <=       #2 rd_addr + 1'b1;
29. end
30.
31. rom_test_top u_rom_test_top(
32.       .rd_clk     ( sys_clk    ),
33.       .rst_n      ( rst_n      ),
34.       .rd_addr    ( rd_addr    ),
35.       .rd_data    ( rd_data    )
36. );
37.
38. endmodule
```

首先在8-15行，是初始化部分。rst_n在最开始被拉低，保持20ns后释放为高电平，同时sys_clk初始化为0。这样在20ns之后，整个电路进入正常工作状态。

接下来17-18行，通过always #10 sys_clk = ~sys_clk;来生成一个周期20ns的时钟信号，即50MHz的系统时钟，作为ROM的读时钟。

核心逻辑在24-29行。在时钟上升沿时，如果复位有效，则rd_addr被清零；如果复位释放，则rd_addr会在每个时钟周期递增1。这里在rd_addr <= #2 rd_addr + 1'b1;中引入了#2 的延时，表示在时钟上升沿过后延迟2ns再更新地址。这通常用于模拟真实硬件中信号到达的延迟，也可以帮助仿真时观察地址和数据之间的关系。

这样设计后，整个testbench的运行过程就是：在复位释放后，ROM的读地址会从0开始，每个时钟周期加1，不断扫描整个地址空间，同时通过端口把ROM中的数据读出到rd_data。

整体功能testbench的主要作用就是**连续递增读ROM的内容**，方便验证ROM初始化的数据是否正确，以及ROM在仿真中的读延迟和数据稳定性。**testbench的逻辑就是通过时钟驱动，让rd_addr递增，从而顺序读出ROM中的全部数据，借助仿真波形检查ROM的读出结果是否符合预期。**

6.5.3. FIFO 仿真测试

```
1.  `timescale 1ns/1ns
2.  module fifo_test_tb();
3.
4.  reg sys_clk;
5.  reg rst_n;
6.
7.  reg             [7:0]   wr_data;
8.  reg                     wr_en;
9.  reg                     rd_en;
10.
11. reg                     rd_state;     //读状态
12. reg                     wr_state;
13.
14. wire            [7:0]   rd_data;
15. reg             [7:0]   rd_cnt;
16.
17. wire            [7:0]   rd_water_level;
18. wire            [7:0]   wr_water_level;
19.
20. initial
21. begin
22.     rst_n   <=  1'd0;
23.     sys_clk <=  1'd0;
24.     #20
25.     rst_n   <=  1'd1;
26.
27.
28. end
29.
30. always#10 sys_clk = ~sys_clk;     //50MHZ
31.
32. always@(posedge sys_clk or negedge rst_n)    begin
33.     if(!rst_n)
34.     begin
35.         wr_state    <=  1'd0;
36.         wr_en   <=  1'd0;
37.         wr_data <=  8'd0;
38.     end
39.     else
40.     begin
41.         case(wr_state)
42.             1'd0:   if(wr_water_level == 127)   //128个数据
```

```
                            begin
                                wr_en       <=      #2 1'd0;
                                wr_data     <=      #2 8'd0;
                                wr_state <=       #2 1'd1;
                            end
                            else
                            begin
                                wr_en       <=      #2 1'd1;
                                wr_data     <=      #2 wr_data+1'b1;
                                wr_state <=       #2 1'd0;
                            end

                1'd1:       if(rd_cnt == 127)
                                wr_state       <=       #2 1'd0;


                default:       wr_state       <=1'd0;
            endcase
        end
    end

    always@(posedge sys_clk or negedge rst_n)     begin
        if(!rst_n)
        begin
            rd_state<=     1'd0;
            rd_en     <=     1'd0;
            rd_cnt    <=     8'd0;
        end
        else
        begin
            case(rd_state)
                1'd0:     if(rd_water_level >= 8'd128)       //等待128个数据
                            begin
                                rd_state       <=       #2 1'd1;
                                rd_en          <=       #2 1'd1;
                            end
                            else
                            begin
                                rd_cnt         <=    #2 8'd0;
                                rd_state       <=     #2 1'd0;
                            end

                1'd1:     begin

                                rd_cnt    <=       #2 rd_cnt + 1'b1;
                                if(rd_cnt == 127)
                                begin
                                    rd_en          <=     #2 1'd0;
                                    rd_state       <=     #2 1'd0;
                                end
                            end
                default:       rd_state       <=     1'd0;
            endcase
        end
    end
```

```
98.
99.   GTP_GRS GRS_INST(
100.      .GRS_N(1'b1)
101.      ) ;
102.
103.  fifo_test_top u_fifo_test_top(
104.      .sys_clk          ( sys_clk          ),
105.      .rst_n            ( rst_n            ),
106.      .wr_data          ( wr_data          ),
107.      .wr_en            ( wr_en            ),
108.      .rd_en            ( rd_en            ),
109.      .wr_water_level   ( wr_water_level   ),
110.      .rd_water_level   ( rd_water_level   ),
111.      .rd_data          ( rd_data          )
112.  );
113.  endmodule
```

涉及到testbench的一些基础操作这里就不再重复，我们关注FIFO的写入与读取逻辑部分。从代码的**32行到62行**和**64行到97行**，分别对应FIFO的写入控制状态机和读取控制状态机。

首先在**初始化部分（20-28行）**，rst_n在20ns时释放，同时生成一个50MHz的系统时钟（30行）。这样在仿真启动后，FIFO可以在复位完成后开始正常工作。

接下来**写入逻辑（32-62行）**。在复位时，wr_state、wr_en和wr_data都清零。复位释放后，进入写状态机：

在**state=0**时，testbench会不断往FIFO中写入数据，每次写入时wr_en拉高，并且wr_data自增。当写入数据达到128个（wr_water_level==127）时，拉低wr_en，清wr_data，并将写状态机切换到state=1。在**state=1**时，写状态机会等待读取计数rd_cnt到达127，才会重新跳转回state=0，继续写数据。这样保证FIFO不会无限写入，而是写满128个数据后进入等待状态，等待读出操作。

然后是**读取逻辑（64-97行）**。在复位时，rd_state、rd_en和rd_cnt被清零。复位释放后，进入读状态机：

在**state=0**时，只有当FIFO中的读水位计数rd_water_level >= 128时，才会启动读取操作，将rd_state切换到state=1，并拉高rd_en。在**state=1**时，开始从FIFO中读取数据，每个周期rd_cnt自增，直到读满128个数据（rd_cnt == 127），再拉低rd_en并回到state=0。如果FIFO数据未达到128，则继续保持等待，直到数据满足条件再开始读出。

这样，写和读状态机配合起来，构成了**写满128个数据→读出128个数据→再次写入→再次读取**的循环过程，保证FIFO的写入和读取可以交替进行。

整体功能实现上，这个testbench的作用就是验证FIFO的写入和读取机制是否正确，尤其是写水位和读水位信号的变化是否符合预期。通过波形可以直观地看到FIFO在写入128个数据后进入读过程，随后又重新开始写入，验证了FIFO的基本功能。testbench通过两个状态机分别控制写和读，每次写满128个数据再读出128个数据，形成一个完整的FIFO功能验证流程。

7.基于紫光 FPGA 的 LED 流水灯

7. 1. 实验简介

实验目的：

通过按键控制4个LED灯按顺序依次点亮和熄灭。OPHW-25开发板有4个用户LED灯（LED0～3），FPGA输出高电平时对应的LED灯亮灯（详情请查看“OPHW-25开发板硬件使用手册”）。控制4个LED灯按顺序依次点亮和熄灭。

实验环境：

Window11

PDS2022.2

硬件环境：

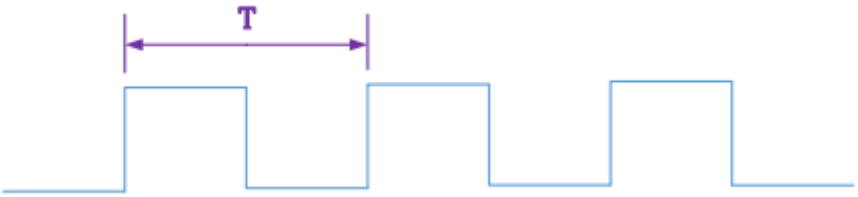
OPHW-25开发板

7. 2. 实验原理

通常的时，分，秒的计时进位大家应该不陌生；

1小时=60分钟=3600秒，当时针转动1小时，秒针跳动3600次；

在数字电路中的时钟信号也是有固定的节奏的，这种节奏的开始到结束的时间,我们通常称之为周期（T）。



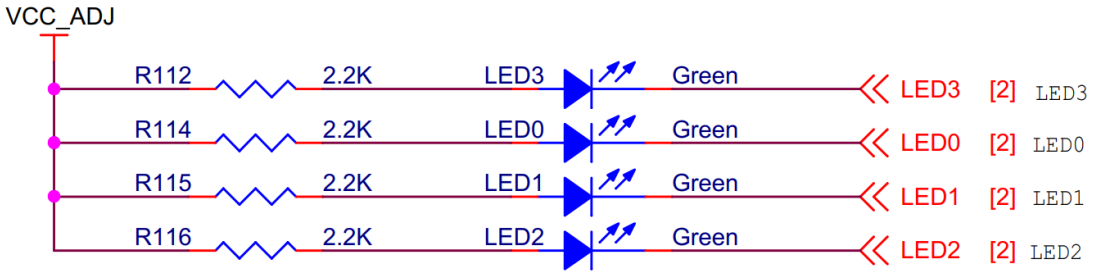
在数字系统中通常关注到时钟的频率,那频率与周期的关系如下：

$$f = \frac{1}{T};$$

OPHW-25板卡上单端时钟有一个50MHz和一个27MHz的晶振提供时钟给到OPHW-25;

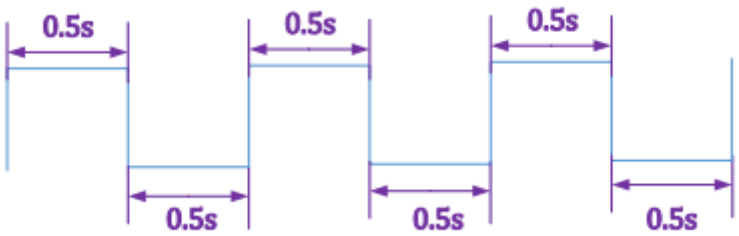
实验分析:

控制LED亮灭需要控制IO输出的高低电平即可(低电平点亮,高电平熄灭),原理图如下:



控制LED依次0.5s亮，0.5s灭,需要控制IO依次输出0.5s高电平，0.5s低电平周期变化。

如下图波形:



若使用50MHz外部输入时钟，时钟周期为20ns（在verilog设计中的计数器的计时原理基本上是一致的，确认输入时钟周期和目标计时时间后可得到计数器的计数值到达多少后可得到计时宽度）；

$$0.5s = 25000000*20ns = 25000000 \times T_{50MHz};$$

IO输出状态只有两种：1或0；我们可以使用一个计数器，计数满25000000个时钟周期时变化不同LED点亮。

7. 3. 实验源码设计

7.3.1.文件头设计

在module之前添加文件头，文件头中包含信息有：公司，作者，时间，设计名，工程名，模块名，目标器件，EDA工具(版本)，模块描述，版本描述（修改描述）等信息；以及仿真时间单位定义；

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:Meyesemi
// Engineer: Will
//
// Create Date: 2023-01-29 20:31
// Design Name:
// Module Name:
// Project Name:
// Target Devices: Pango
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 1.0 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

`define UD #1
```

`timescale1ns/1ps表示仿真精度是1ns，显示精度是1ps；

`defineUD#1定义UD表示#1；#1仅仿真有效，表示延时一个仿真精度，结合上一条语句表示延时1ns；

7.3.2.设计 module

```
module led_test(
input                clk,
input                rstn,

output [3:0]     led
);
```

此段代码是标准的module创建的模型，module创建时需要确认输入输出信号并定义好位宽，之后在对module进行具体的逻辑设计；管脚与管脚之间用“,”隔开，最后一个管脚不用间隔符号；

创建module时需要定义输入输出信号;本实验输入时钟和复位即可，输出是控制LED的亮灭，OPHW-25板卡上共有4个LED，故而输出4bit位宽的信号；

单个状态计数25_000_000，即24_999_999=25'b1_0111_1101_0111_1000_0011_1111;所以计数器的位宽为25位即可；

```
//time counter
    always @(posedge clk)
    begin
        if(!rstn)
            led_light_cnt <= `UD 26'd0;
        else if(led_light_cnt == 26'd24_999_999)
            led_light_cnt <= `UD 26'd0;
        else
            led_light_cnt <= `UD led_light_cnt + 26'd1;
        end

```

当计数器计数到25'd24_999_999时，计数过程包含了从0～26'd2499_9999的时钟周期，故而总时长时25'd25_000_000×Tclk;硬件输入时钟为50MHz,所以此计数器的计数周期是0.5s;

在指定的时间刻度上对LED的状态进行变更，以达到控制LED依次亮灭的目的；led_light_cnt的计时周期为0.5s，故在led_light_cnt上取一个点来变更LED的显示状态即可完成每隔0.5s让LED显示发生变化；由于LED亮和灭只有两个状态，在赋值处理上将寄存器进行移位操作；

```
//led status change
    always @(posedge clk)
    begin
        if(!rstn)
            led_status <= `UD 4'b0001;
        else if(led_light_cnt == 25'd24_999_999)
            led_status <= `UD {led_status[2:0],led_status[3]};
    end

    assign led = led_status;
```

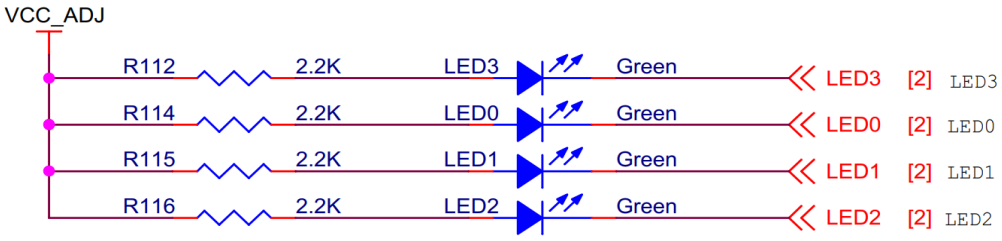
7.3.3.完整的 Module

```
module led_test(
    input               clk,
    input               rstn,

    output [3:0]    led
);

//=============================================================================
//reg and wire

    reg [25:0] led_light_cnt    = 26'd0             ;
    reg [ 7:0] led_status       = 4'b0001   ;

//time counter
    always @(posedge clk)
    begin
        if(!rstn)
            led_light_cnt <= `UD 26'd0;
```

```
            else if(led_light_cnt == 26'd24_999_999)
                led_light_cnt <= `UD 26'd0;
            else
                led_light_cnt <= `UD led_light_cnt + 26'd1;
        end

//led status change
        always @(posedge clk)
        begin
            if(!rstn)
                led_status <= `UD 8'b0000_0001;
            else if(led_light_cnt == 25'd24_999_999)
                led_status <= `UD {led_status[2:0],led_status[3]};
        end

        assign led = led_status;

endmodule
```

7.3.4.硬件管脚分配

OPHW-25的LED和CLK与FPGA的IO连接部分的原理图如下，详情可查看硬件使用手册或原理图：



信号	OPHW-25Pin
LED0	E18
LED1	F17
LED2	H18
LED3	H14

复位设计是低电平有效，OPHW-25开发板提供了4个用户按键，按键低电平有效，但按键按下时，IO上的输入电压为低；当没有按下按键时，IO上的输入电压为高电平；选择任一个用户按键作为复位输入即可。

7. 4. 实验现象

4颗LED灯按照设定的顺序和时间依次点亮和熄灭

8.基于紫光 FPGA 的键控流水灯实验例程

8. 1. 实验简介

实验目的：

OPHW-25开发板有4个用户LED灯，FPGA输出低电平时对应的LED灯亮灯（详情请查看“OPHW-25开发板硬件使用手册”）。由USER_BUTTON1按键输入，切换USER_LED0~USER_LED3的输出效果。

实验环境：

Window11

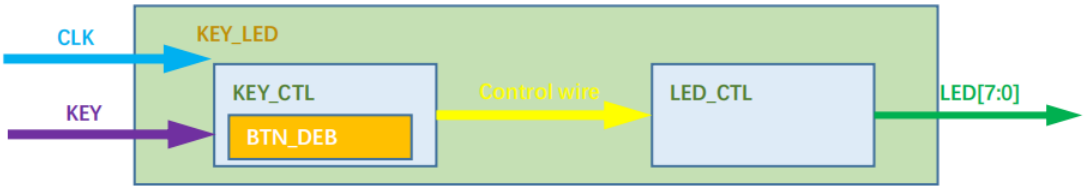
PDS2022.2

硬件环境：

OPHW-25开发板

8. 2. 实验原理

实现框架如下：



（1）顶层实现按键切换LED的流水灯状态；

（2）需要设计一个输入控制模块及一个输出控制模块；

这个实验带大家将多个模块整合成为一个工程，涉及到的知识点有子模块设计、模块例化；子模块的设计主要是依据功能定位，确定输入输出，再做具体的设计；

模块例化方式如下：

```
module_name #   (
.PARAM              (       PARAM_SET )
// PARAM为例化模块的常量接口；PARAM_SET为常量赋值内容
) unint_name(
// module_name   为例化module名；unint_name为例化后单元名称
.port                   (       signal                  )
 // port为例化模块中的管脚；  signal为当前模块的信号
);
```

8.2.1.按键控制模块功能

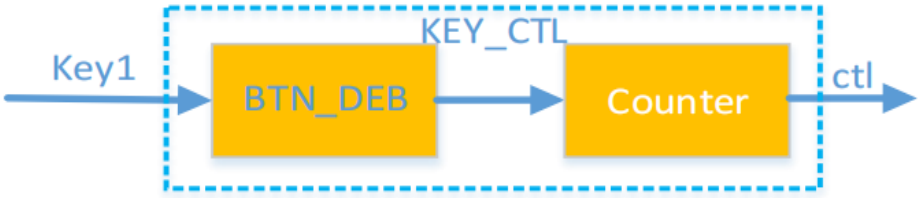
接收按键输入信号。统计按键按下次数，由于流水灯模式是3种，计数统计范围是0～2循环，将计数结果传递给LED控制模块；

根据需求输入信号有：时钟，按键；输出信号有：流水灯控制信号；

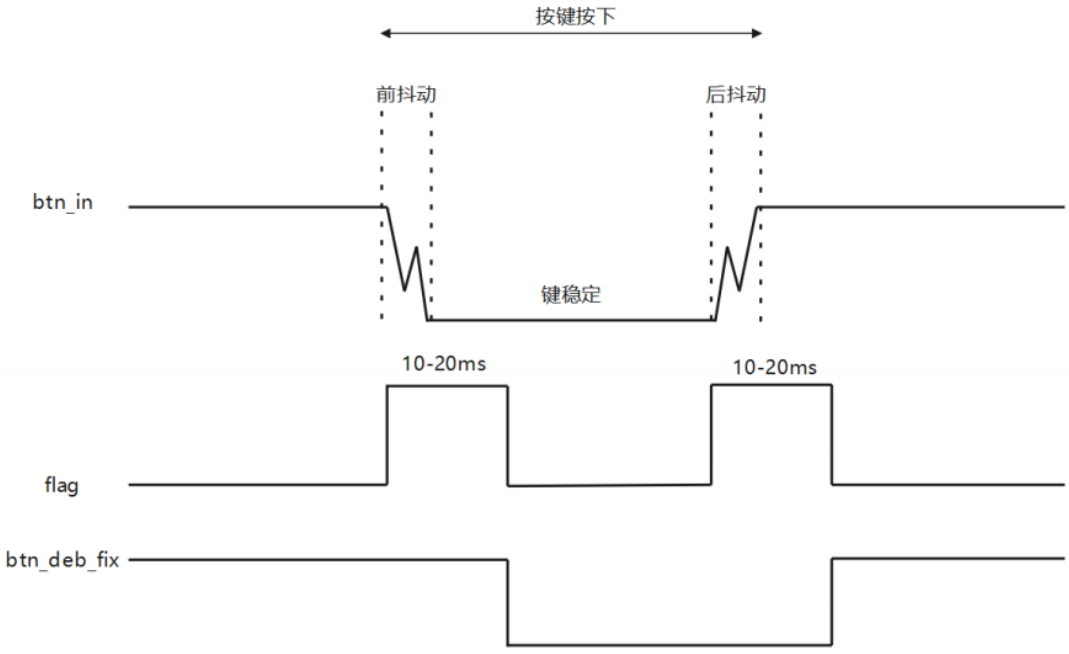
内部功能处理：

<1>内部需要对按键信号做消抖处理；

<2>按键触发计数器（计数值输出）改变继而调整流水灯的状态；



8.2.2.按键消抖模块



前后抖动时间约为5～10ms，取按键抖动区间开始标识，持续10-20ms后标识归零，在抖动区间内输出保持，非消抖区间，按键状态输出。

8.2.3.LED 控制模块功能

3种流水灯模式有按键传递过来的计数控制切换，每一个LED的显示状态完整后进入下一模式初始化。根据需求可得到如下信息：

输入信号：时钟，流水灯模式控制信号；出信号：8bit位宽的LED控制信号；功能处理注意事项：流水灯状态切换点，不同状态的切换时如何初始化；

8. 3. 实验源码设计

8.3.1.顶层文件源码

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module key_led_top(
4.      input               clk,    //50MHz
5.      input               key,
6.
7.      output [3:0]    led
8.  );
9.
10.     wire [1:0] ctrl;
11.
12.     key_ctl key_ctl(
13.         .clk        (  clk  ),  //input             clk,
14.         .key        (  key  ),  //input             key,
15.
16.         .ctrl       (  ctrl  )  //output        [1:0] ctrl
17.     );
18.
19.     led u_led(
20.         .clk    (  clk    ),      //input             clk,
21.         .ctrl   (  ctrl   ),      //input   [1:0]   ctrl,
22.
23.         .led    (  led    )     //output [7:0]   led
24.     );
25.
26. endmodule
```

8.3.2.按键控制模块

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module key_ctl(
4.      input               clk,
5.      input               key,
6.
7.      output      [1:0] ctrl
8.  );
9.
10.     wire btn_deb;
11.
```

```
12.     btn_deb_fix#(
13.         .BTN_WIDTH    (   4'd1            ), //parameter                       BTN_WIDTH = 4'd8
14.         .BTN_DELAY    (20'h7_ffff      )
15.     ) u_btn_deb
16.     (
17.         .clk            (   clk             ),//input                                 clk,
18.         .btn_in         (   key             ),//input         [BTN_WIDTH-1:0] btn_in,
19.
20.         .btn_deb_fix (   btn_deb       ) //output reg [BTN_WIDTH-1:0] btn_deb
21.     );
22.
23.     reg btn_deb_1d;
24.     always @(posedge clk)
25.     begin
26.         btn_deb_1d <= `UD btn_deb;
27.     end
28.
29.     reg [1:0]   key_push_cnt=2'd0;
30.     always @(posedge clk)
31.     begin
32.         if(~btn_deb & btn_deb_1d)
33.         begin
34.             if(key_push_cnt == 2'd2)
35.                 key_push_cnt <= `UD 2'd0;
36.             else
37.                 key_push_cnt <= `UD key_push_cnt + 2'd1;
38.         end
39.     end
40.
41.     assign ctrl = key_push_cnt;
42.
43. endmodule
```

整个模块主要功能是对外部输入的按键信号进行消抖，并通过计数器实现按键多次按下时的模式切换控制。

在10-21行，这里实例化了一个btn_deb_fix模块，对输入key进行消抖处理。由于机械按键在按下和释放时会产生抖动，所以必须经过消抖电路，输出稳定的btn_deb信号。参数设置了单通道按键（BTN_WIDTH=1）和延迟阈值（BTN_DELAY=20'h7_ffff），确保按键信号稳定后才有效。

在23-27行，通过一个时序寄存器btn_deb_1d保存消抖信号的一个时钟周期延迟。这样可以配合当前周期的btn_deb，用来检测按键的边沿变化。

重点逻辑在29-39行：定义了一个2位寄存器key_push_cnt，用来计数按键的按下次数。在always块中，通过条件if(~btn_deb&btn_deb_1d)检测到**按键从按下到释放的上升沿**（即一次有效的按键动作）。每次检测到有效按键动作时，计数器key_push_cnt+1。计数器达到2'd2时，再次按键会将其清零，形成一个**0→1→2→0**的循环计数。

最后在41行，将key_push_cnt的值输出到ctrl。这样ctrl就会随着按键次数循环变化，通常可用来切换工作模式，**按一次进入模式1，按两次进入模式2，再按一次回到模式0。**

总体说btn_deb_fix模块解决了机械按键抖动问题；边沿检测逻辑确保每次按键只触发一次计数；2位计数器实现了多模式循环切换功能；输出ctrl作为最终控制信号，直接驱动其他模块。

8.3.3.按键消抖模块

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module btn_deb_fix#(
4.      parameter            BTN_WIDTH = 4'd8,
5.      parameter            BTN_DELAY = 20'h7_ffff
6.  )
7.  (
8.      input                                clk,   //
9.      input        [BTN_WIDTH-1:0] btn_in,
10.
11.     output reg [BTN_WIDTH-1:0] btn_deb_fix
12. );
13.
14.     //16'h3ad43;
15.     reg [19:0]              cnt[BTN_WIDTH-1:0];
16.     reg [BTN_WIDTH-1:0] flag;
17.
18.     reg [BTN_WIDTH-1:0] btn_in_reg;
19.
20.     always @(posedge clk)
21.     begin
22.       btn_in_reg <= `UD btn_in;
23.     end
24.
25.     genvar i;
26.     generate
27.         for(i=0;i<BTN_WIDTH;i=i+1)
28.         begin
29.             always @(posedge clk)
30.             begin
31.               if (btn_in_reg[i] ^ btn_in[i]) //取按键边沿开始抖动区间标识
32.                 flag[i] <= `UD 1'b1;
33.               else if (cnt[i]==BTN_DELAY)       //持续10ms-20ms后归零
34.                 flag[i] <= `UD 1'b0;
35.                   else
36.                       flag[i] <= `UD flag[i];
37.             end
38.
39.             always @(posedge clk)
40.             begin
41.               if(cnt[i]==BTN_DELAY)           //计数10ms-20ms时归零
42.                 cnt[i] <= `UD 20'd0;
43.               else if(flag[i])                  //抖动区间有效时计数
44.                 cnt[i] <= `UD cnt[i] + 1'b1;
45.               else                                    //非抖动区间保持0
46.                 cnt[i] <= `UD 20'd0;
47.             end
```

```
48.
49.                 always @(posedge clk)
50.                 begin
51.                  if(flag[i])                    //抖动区间，消抖输出保持
52.                    btn_deb_fix[i] <= `UD btn_deb_fix[i];
53.                  else                             //非抖动区间，按键状态传递到消抖输出
54.                    btn_deb_fix[i] <= `UD btn_in[i];
55.                 end
56.             end
57.     endgenerate
58.
59. endmodule
```

代码的25行到57行是按键消抖的核心实现，主要用于对多路机械按键信号进行消抖，输出稳定的按键状态。这里只讲解主要实现的功能。首先代码的20-23行，通过一个时钟上升沿触发的always块，将输入按键状态寄存到btn_in_reg，这是为了后续检测按键边沿变化，保证时序逻辑能够正确捕捉按键的跳变。接着在生成块中，对于每一路按键，都存在三个always块，分别用于抖动检测、计数器累加和消抖输出。

代码的31-37行是抖动检测逻辑。当上一周期的按键状态btn_in_reg[i]与当前按键状态btn_in[i]不一致时，说明检测到按键边沿变化，于是将抖动标志flag[i]拉高，表示当前按键进入抖动阶段；如果计数器cnt[i]已经达到设定的消抖时间BTN_DELAY，则将flag[i]清零；否则保持原状态不变。这个逻辑保证了按键在刚触发或释放的瞬间进入抖动阶段，抖动阶段持续时间由计数器控制。

代码的41-47行是计数器逻辑。在抖动阶段，计数器cnt[i]每个时钟周期累加，用于记录抖动持续时间，当计数器达到设定的消抖时间时自动清零。如果按键不在抖动阶段，计数器保持0。这一逻辑的作用是为抖动检测提供时间窗口，从而滤除机械按键可能存在的瞬时跳变。

代码的51-55行是消抖输出逻辑。当按键处于抖动阶段时，消抖输出btn_deb_fix[i]保持原状态不变，避免在抖动过程中出现不稳定输出；而在非抖动阶段，按键的实际状态直接传递到消抖输出，从而生成稳定的按键信号。这里需要注意，由于是时序逻辑，赋值总在下一个时钟周期才生效，所以在抖动开始或结束的边沿，输出信号会在下一个时钟周期才更新。时序逻辑的赋值总在下一个时钟周期生效，所以抖动开始或结束时的操作会在当前时钟周期观察到的输出上有延迟，但不会影响整体的消抖效果。

8.3.4.LED 控制模块

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module led(
4.      input              clk,//50MHz
5.      input   [1:0]   ctrl,
6.
7.      output [3:0]   led
8.  );
```

```
9.
10.     reg [24:0] led_light_cnt = 25'd0;
11.     reg [ 3:0] led_status = 4'b1000;
12.
13.     //   time counter
14.     always @(posedge clk)
15.     begin
16.         if(led_light_cnt == 25'd24_999_999)
17.             led_light_cnt <= `UD 25'd0;
18.         else
19.             led_light_cnt <= `UD led_light_cnt + 25'd1;
20.     end
21.
22.     reg [1:0] ctrl_1d=0;       //保存上一个led状态周期的ctrl值
23.     always @(posedge clk)
24.     begin
25.         if(led_light_cnt == 25'd0)
26.             ctrl_1d <= ctrl;
27.     end
28.
29.     // led status change
30.     always @(posedge clk)
31.     begin
32.         if(led_light_cnt == 25'd24_999_999)//0.5s  周期
33.         begin
34.             case(ctrl)
35.                 2'd0 :   //从高位到低位的led流水灯
36.                 begin
37.                     if(ctrl_1d != ctrl)
38.                         led_status <= `UD 4'b1000;
39.                     else
40.                         led_status <= `UD {led_status[0],led_status[3:1]};
41.                 end
42.                 2'd1 :   //隔一亮一交替
43.                 begin
44.                     if(ctrl_1d != ctrl)
45.                         led_status <= `UD 4'b1010;
46.                     else
47.                         led_status <= `UD ~led_status;
48.                 end
49.                 2'd2 :   //从高位到低位暗灯流水
50.                 begin
51.                     if(ctrl_1d != ctrl )
52.                         led_status <= `UD 4'b0111;
53.                     else
54.                         led_status <= `UD {led_status[0],led_status[3:1]};
55.                 end
56.             endcase
57.         end
58.     end
59.
60.     assign led = led_status;
61.
62. endmodule
```

从代码的14行到58行是LED的闪烁控制逻辑，主要用于根据输入的控制信号ctrl实现不同模式的LED灯显示。这里只讲解主要实现的功能。首先代码的14-20行是时间计数器逻辑，通过一个50MHz时钟对led_light_cnt计数，每个时钟上升沿计数器加1，当计数器达到24,999,999时归零，这样形成了一个0.5秒的周期，用作LED模式切换的时间基准。

代码的23-27行用于保存上一个LED状态周期的控制信号值ctrl_1d，在每个计数器归零的时刻，将当前的ctrl保存到ctrl_1d中，以便在模式切换时判断当前周期是否与上一个周期不同，确保在切换模式时能够正确初始化LED状态。

代码的30-58行是LED状态变化逻辑。在每个计数器达到24,999,999，即0.5秒时，进入一个case分支，根据ctrl信号选择不同的LED显示模式。当ctrl=2'd0时，实现从高位到低位的LED流水灯，如果模式刚刚切换（ctrl_1d != ctrl），则将LED状态初始化为8'b1000_0000，否则每个周期将LED状态循环右移一位，形成流水灯效果。当ctrl=2'd1时，实现隔一亮一交替的LED闪烁，如果模式切换则初始化为8'b1010_1010，否则每个周期取反LED状态，实现交替闪烁效果。当ctrl=2'd2时，实现从高位到低位的暗灯流水效果，切换模式时初始化为8'b0111_1111，否则同样通过右移循环实现流水效果。

assign 语句 assign led = led_status;将计算得到的LED状态输出到模块端口led，使外部能够看到LED灯的显示效果。不同模式下LED状态会按0.5秒周期循环变化。这个模块通过计数器实现时间基准，通过保存上一个控制周期的信号判断模式切换，并根据不同模式更新LED状态，实现从流水灯、交替闪烁到暗灯流水的多种LED显示效果

8.4. 实验现象

每按下一次KEY1，LED灯状态切换一次，总共三种LED模式供循环切换；

LED模式一：从高位到低位的LED流水灯；LED模式二：隔一亮一交替点亮；

LED模式三：从高位到低位暗灯流水；

9.基于紫光 FPGA 的 UART 串口通信

9. 1. 实验简介

实验目的：

OPHW-25开发板集成了一路USB转串口模块，采用的USB-UART芯片CP2102,USB接口采用USBTypeC接口，可以用一根USBTypeC线连接到PC的USB口进行串口数据通信（详情请查看“OPHW-25开发板硬件使用手册”）。通过本实验实现FPGA与PC之间的串口收发实验。串口通信时波特率设置为115200bps，数据格式为1位起始位、8位数据位、无校验位、1位结束位。板子1s向串口助手发送一次十进制显示的“www.meyesemi.com”，通过串口助手向板子以十六进制形式发送数字（00~FF），LED以二进制显示亮起。

实验环境：

Window11

PDS2022.2

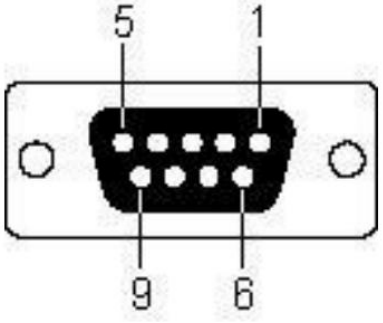
硬件环境：

OPHW-25开发板

9. 2. 实验原理

9.2.1.串口原理

从图 8.2-1我们可以看到标准串口接口是 9 根线，具体含义如下：



数据线：

TXD（pin 3）： 串口数据输出(Transmit Data)

RXD（pin 2）： 串口数据输入(Receive Data)

握手：

RTS（pin 7）： 发送数据请求(Request to Send)

CTS（pin 8）：清除发送(Clear to Send)

DSR（pin 6）：数据发送就绪(Data Send Ready)

DCD（pin 1）：数据载波检测(Data Carrier Detect)

DTR（pin 4）：数据终端就绪(Data Terminal Ready)

地线：

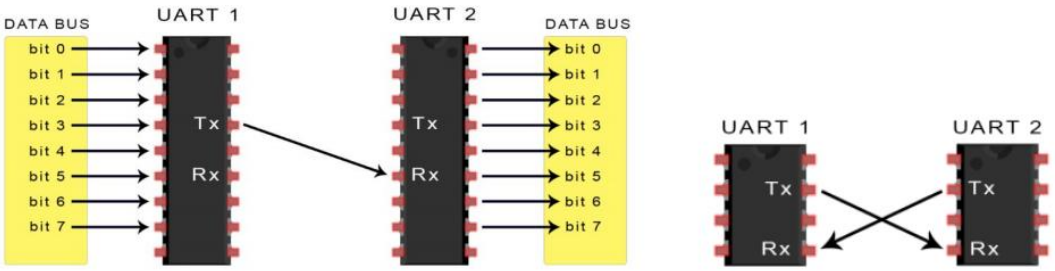
GND（pin 5）：地线

其它：

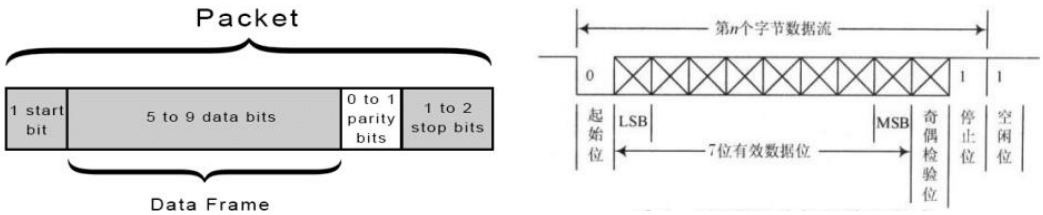
RI（pin 9）：铃声指示

通常我们用 RS232 串口仅用到了9根传输线中的三根：TXD，RXD，GND。但是对于数据传输，双方必须对数据传输采用使用相同的波特率，约定同样的传输模式（传输架构，握手条件等）。尽管这种方法对于大多数应用已经足够，但是对于接收方过载的情况这种使用受到限制。

RS232 的串口连接方式：



串口传输协议如下：



起始位：先发出一个逻辑”0”信号，表示传输字符的开始。

数据位：可以是5~8位逻辑”0”或”1”。如ASCII码（7位），扩展BCD码（8 位）。LSB表示低位，MSB表示高位，有效数据的传输顺序为低位在前高位在后。

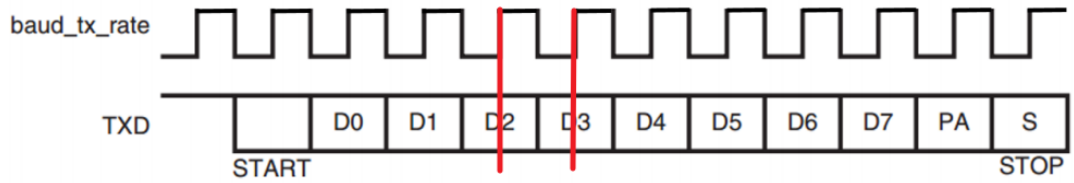
校验位：数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)。

停止位：它是一个字符数据的结束标志。可以是1位、1.5位、2位的高电平。

空闲位：处于逻辑“1”状态，表示当前线路上没有资料传送。

波特率：uart 中的波特率就可以认为是比特率，即每秒传输的位数(bit)。一般选波特率都会有9600,19200,115200等选项。其实意思就是每秒传输这么多个比特位数(bit)。

引入波特率的概念后可得到串口的传输节奏如下：



细心的话可以发现数据的传输都是在数据稳定后的中心时刻，接收数据其实也是，都是在中心时刻采集数据，此时的数据是最稳定的。

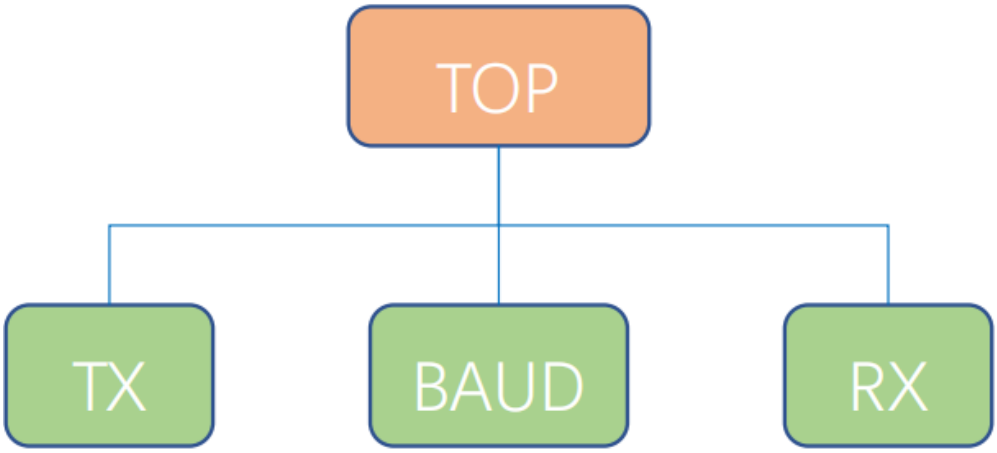
9.2.2.串口发送字符

从前面串口协议中可以了解到串口每次传输可以以有5～8bit数据，在计算机中字符通常用ASCII码（7bit）表示，所以字符的发送可以用ASCII码发送。查询ASCII码表格可得到：“www.meyesemi.com”用到的字符对应ASCII码；

```
8'h1  : write_data <= `UD 8'h77;// ASCII code is w
8'h2  : write_data <= `UD 8'h77;// ASCII code is w
8'h3  : write_data <= `UD 8'h77;// ASCII code is w
8'h4  : write_data <= `UD 8'h2E;// ASCII code is .
8'h5  : write_data <= `UD 8'h6D;// ASCII code is m
8'h6  : write_data <= `UD 8'h65;// ASCII code is e
8'h7  : write_data <= `UD 8'h79;// ASCII code is y
8'h8  : write_data <= `UD 8'h65;// ASCII code is e
8'h9  : write_data <= `UD 8'h73;// ASCII code is s
8'ha  : write_data <= `UD 8'h65;// ASCII code is e
8'hb  : write_data <= `UD 8'h6D;// ASCII code is m
8'hc  : write_data <= `UD 8'h69;// ASCII code is i
8'hd  : write_data <= `UD 8'h2E;// ASCII code is .
8'he  : write_data <= `UD 8'h63;// ASCII code is c
8'hf  : write_data <= `UD 8'h6F;// ASCII code is o
8'h10 : write_data <= `UD 8'h6D;// ASCII code is m
```

9. 3. 实验源码设计

从实验目的分析可将实验做如下划分：



从原理上分析波特率的计算是一个计数器，发射和接收可复用，我们在设计时为保持TX，或RX的完整性，故将波特周期计数器集成在各自模块内部；

上述分析仅仅搭建好OPHW-25的与PC通信的桥梁UART，传输的数据没有体现。故而需要增加发送数据模块，与接收数据模块；

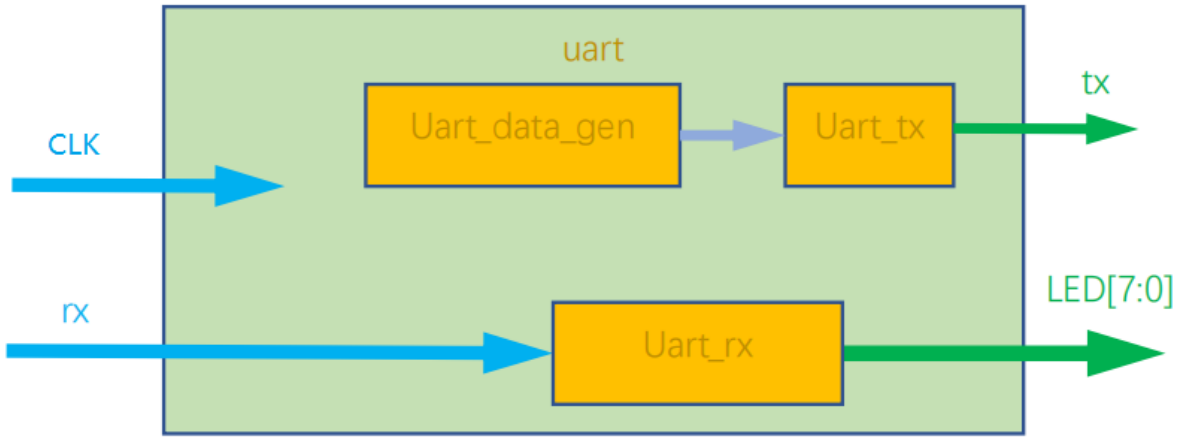


图 8.4-2

9.3.1.串口发送模块设计

目标：接收到一个发送命令信号时，将data[7:0]->依次发出{start,data[0:7],stop}共10bit数据（无校验位，停止位1bit）；

有两种方法可以将一个并行数据串行化；

方法一：通过bit计数与baud计数控制移位输出；

```
1.  // transmit bit
2.          always@(posedge clk)
3.          begin
4.              if(!rstn)
```

```
5.                  txd <= `UD 1'b1;
6.              else
7.              begin
8.                  if(trans_en)
9.                  Begin
10.  //   将开始标志和停止标志以及传输数据集成放到trans_data 中可用下方语句
11.  //                  txd <= `UD trans_data[trans_bit];
12.  // 单 bit 控制用下方语句
13.                      case(trans_bit)
14.                          4'h0       :txd <= `UD 1'b0;
15.                           4'h1       :txd <= `UD tx_data_reg[0];
16.                           4'h2       :txd <= `UD tx_data_reg[1];
17.                           4'h3       :txd <= `UD tx_data_reg[2];
18.                           4'h4       :txd <= `UD tx_data_reg[3];
19.                           4'h5       :txd <= `UD tx_data_reg[4];
20.                           4'h6       :txd <= `UD tx_data_reg[5];
21.                           4'h7       :txd <= `UD tx_data_reg[6];
22.                           4'h8       :txd <= `UD tx_data_reg[7];
23.                          4'h9       :txd <= `UD 1'b1;
24.                           default :txd <= `UD 1'b1;
25.                      endcase
26.                 end
27.                 else
28.                     txd <= `UD 1'b1;
29.             end
30.         end
```

这段代码用于实现 UART 模块中每一位数据的发送逻辑。模块通过一个时钟同步控制信号 txd 的高低电平，从而实现串行数据的输出。

首先看第2-30行的 always 块逻辑，这是在每个时钟上升沿触发的时序逻辑。模块首先判断复位信号 rstn，如果复位有效，txd 被置为高电平 1'b1，保持串口空闲状态。因为 UART 总是空闲高电平，复位时也要保证输出正确。

当复位结束后，如果传输使能信号 trans_en 有效，模块进入数据传输状态。代码通过 case 语句根据 trans_bit 的值选择发送哪一位。具体来说，trans_bit 从 0 到 9 分别对应 UART 串口的起始位、8 位数据位和停止位：当 trans_bit=0 时，发送起始位 0，表示数据传输开始；

当 trans_bit=1 到 8 时，依次发送数据寄存器 tx_data_reg[0] 到 tx_data_reg[7] 的数据位，按照从低位到高位顺序传输；当 trans_bit=9 时，发送停止位 1，表示数据传输结束；默认情况下，也保持高电平 1，保证空闲状态输出正确。

如果传输使能 trans_en 为无效状态，模块将 txd 保持高电平，这样可以保证 UART 总是在空闲时输出逻辑 1。

方法二：通过bit计数与baud计数控制状态跳转，在状态机中输出；

```
1.  //  logical ouput  状态机输出
2.      always @ (posedge clk)
3.      begin
4.          if(tx_en)
5.          begin
6.              case(tx_state)
```

```
                IDLE        :   uart_tx   <= `UD 1'h1;   //空闲状态输出高电平
                SEND_START :   uart_tx <= `UD 1'h0; //start  状态发送一个波特周期的低电平
                SEND_DATA   :                          //发送状态每个波特周期发送一个  bit;
              begin
                  case(tx_bit_cnt)
                      3'h0   :   uart_tx   <= `UD trans_data[0];
                      3'h1   :   uart_tx   <= `UD trans_data[1];
                      3'h2   :   uart_tx   <= `UD trans_data[2];
                      3'h3   :   uart_tx   <= `UD trans_data[3];
                      3'h4   :   uart_tx   <= `UD trans_data[4];
                      3'h5   :   uart_tx   <= `UD trans_data[5];
                      3'h6   :   uart_tx   <= `UD trans_data[6];
                      3'h7   :   uart_tx   <= `UD trans_data[7];
                      default: uart_tx   <= `UD 1'h1;
                  endcase
               end
              SEND_STOP   :   uart_tx   <= `UD 1'h1; //发送停止状态  输出  1  个波特周期高电平
              default      :   uart_tx   <= `UD 1'h1; //  其他状态默认与空闲状态一致，保持高电平输出
          endcase
      end
      else
          uart_tx <= `UD 1'h1;
  end
这里笔者采用方法二，完整 module 设计如下：
`timescale 1ns / 1ps
`define UD #1

module uart_tx #(
    parameter                BPS_NUM   =      16'd434
//  设置波特率为 4800 时，bit 位宽时钟周期个数:50MHz set 10417   40MHz set 8333
//  设置波特率为 9600 时，bit 位宽时钟周期个数:50MHz set 5208    40MHz set 4167
//  设置波特率为 115200 时，bit 位宽时钟周期个数:50MHz set 434   40MHz set 347 12M set 104
)
(
    input              clk,             // clock                                          时钟信号
    input [7:0]      tx_data,        // uart tx data signal byte;                    等待发送的字节数据
    input              tx_pluse,       // uart tx enable signal,rising is active;  发送模块发送触发信号

    output reg       uart_tx,        // uart tx transmit data line                   发送模块串口发送信号线
    output             tx_busy          // uart tx module work states,high is busy;发送模块忙状态指示
);

    //=============================================================================
    //wire and reg in the module
    //=============================================================================
    reg                  tx_pluse_reg =0;

    reg [2:0]       tx_bit_cnt=0; //the bits number has transmited.

    reg [2:0]       tx_state=0;    //current state of tx state machine.
    reg [2:0]       tx_state_n=0; //next state of tx state machine.

    reg [3:0]       pluse_delay_cnt=0;
    reg                  tx_en = 0;
```

```
62.     // uart tx state machine's state
63.     localparam  IDLE      = 4'h0; //tx state machine's state.空闲状态
64.     localparam  SEND_START = 4'h1; //tx state machine's state.发送 start 状态
65.     localparam  SEND_DATA  = 4'h2; //tx state machine's state.发送数据状态
66.     localparam  SEND_STOP  = 4'h3; //tx state machine's state.发送 stop 状态
67.     localparam  SEND_END   = 4'h4; //tx state machine's state.发送结束状态
68.
69.     // uart bps set   the clk's frequency is 50MHz
70.     reg [15:0]    clk_div_cnt=0; //count for division the clock.
71.
72.     //=============================================================================
73.     //logic
74.     //=============================================================================
75.     assign tx_busy = (tx_state != IDLE);
76.     //some control single.
77.
78.     always @(posedge clk)
79.     begin
80.         tx_pluse_reg <= `UD tx_pluse;
81.     end
82.
83.     // uart  模块发送工作使能标志信号
84.     always @(posedge clk)
85.     begin
86.         if(~tx_pluse_reg & tx_pluse)
87.             tx_en <= 1'b1;
88.         else if(tx_state == SEND_END)
89.             tx_en <= 1'b0;
90.     end
91.
92.     //division the clock to satisfy baud rate.波特周期计数器
93.     always @ (posedge clk)
94.     begin
95.         if(clk_div_cnt == BPS_NUM || (~tx_pluse_reg & tx_pluse))
96.             clk_div_cnt     <= `UD 16'h0;
97.         else
98.             clk_div_cnt     <= `UD clk_div_cnt + 16'h1;
99.     end
100.
101.    //count the number has transmited.发送数据状态中，发送 bit 位计数，以波特周期累加
102.    always @ (posedge clk)
103.    begin
104.        if(!tx_en)
105.            tx_bit_cnt      <= `UD 3'h0;
106.        else if((tx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
107.            tx_bit_cnt      <= `UD 3'h0;
108.        else if((tx_state == SEND_DATA) && (clk_div_cnt == BPS_NUM))
109.            tx_bit_cnt      <= `UD tx_bit_cnt + 3'h1;
110.        else
111.            tx_bit_cnt      <= `UD tx_bit_cnt;
112.    end
113.
114.    //=============================================================================
115.    //transmitter state machine
116.    //=============================================================================
```

```
117.
118.     //     state change  状态跳转
119.     always @(posedge clk)
120.     begin
121.         tx_state <= tx_state_n;
122.     end
123.
124.     //     state change condition  状态跳转条件及规律
125.     always @ (*)
126.     begin
127.       case(tx_state)
128.         IDLE       :
129.         begin
130.        //检测到  tx_pluse  上升沿后，立即进入  SEND_START  状态
131.                     if(~tx_pluse_reg & tx_pluse)
132.                       tx_state_n = SEND_START;
133.                  else
134.                       tx_state_n = tx_state;
135.           end
136.               SEND_START :
137.               Begin
138.                //发送一个波特周期的低电平后进入，发送数据状态
139.                     if(clk_div_cnt == BPS_NUM)
140.                       tx_state_n = SEND_DATA;
141.                 else
142.                   tx_state_n = tx_state;
143.           end
144.               SEND_DATA :
145.               Begin
146.                //计时 8 个波特周期（发送了 8bit 数据）
147.                     if(tx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
148.                       tx_state_n = SEND_STOP;                          //跳转到发送 stop 状态
149.                 else
150.                   tx_state_n = tx_state;
151.           end
152.               SEND_STOP :
153.               begin
154.                     if(clk_div_cnt == BPS_NUM)              //设置停止位宽为 1 个波特周期计数
155.                  //发送一个波特周期的高电平，之后跳转到发送结束状态
156.                       tx_state_n = SEND_END;
157.                 else
158.                   tx_state_n = tx_state;
159.               end
160.               SEND_END :    tx_state_n = IDLE;
161.               default :    tx_state_n = IDLE;
162.             endcase
163.         end
164.
165.         //     logical ouput   状态机输出
166.         always @ (posedge clk)
167.         begin
168.           if(tx_en)
169.           begin
170.                case(tx_state)
171.                     IDLE          :    uart_tx    <= `UD 1'h1;              //空闲状态输出高电平
```

```
172.                    SEND_START :   uart_tx   <= `UD 1'h0;   //start 状态发送一个波特周期的低电平
173.                    SEND_DATA  :            //发送状态每个波特周期发送一个 bit;
174.                    begin
175.                        case(tx_bit_cnt)
176.                            3'h0   :   uart_tx   <= `UD tx_data[0];
177.                            3'h1   :   uart_tx   <= `UD tx_data[1];
178.                            3'h2   :   uart_tx   <= `UD tx_data[2];
179.                            3'h3   :   uart_tx   <= `UD tx_data[3];
180.                            3'h4   :   uart_tx   <= `UD tx_data[4];
181.                            3'h5   :   uart_tx   <= `UD tx_data[5];
182.                            3'h6   :   uart_tx   <= `UD tx_data[6];
183.                            3'h7   :   uart_tx   <= `UD tx_data[7];
184.                            default: uart_tx   <= `UD 1'h1;
185.                        endcase
186.                    end
187.                //发送停止状态 输出 1 个波特周期高电平
188.                    SEND_STOP  :   uart_tx   <= `UD 1'h1;
189.
190.                    default    :   uart_tx   <= `UD 1'h1;
191.                // 其他状态默认与空闲状态一致，保持高电平输出
192.                endcase
193.            end
194.            else
195.                uart_tx <= `UD 1'h1;
196.        end
197.
198.        endmodule
```

模块代码使用状态机实现了 UART 串口的字节发送功能，通过波特率控制时钟分频和状态机管理数据传输流程。模块以 50MHz 时钟为基础，通过参数 BPS_NUM 配置波特率的时钟周期数，例如 4800 波特率对应 10417 个时钟周期，9600 波特率对应 5208 个时钟周期，115200 波特率对应 434 个时钟周期。

代码的48行到166行是 UART 串口发送控制逻辑，主要用于将输入的字节数据 tx_data 按照设定的波特率发送到串口输出 uart_tx，并通过 tx_busy 输出模块忙状态指示。

首先代码的48-51行是 tx_pluse 同步逻辑，通过寄存器 tx_pluse_reg 在每个时钟上升沿采样 tx_pluse 信号，用于检测发送触发的上升沿。紧接着在代码的54-60行，通过判断 tx_pluse 上升沿设置发送使能 tx_en，当检测到触发信号时拉高 tx_en，表示模块开始发送；当状态机进入发送结束状态 SEND_END 时，将 tx_en 清零，模块返回空闲状态。

代码的63-69行是波特率周期计数器 clk_div_cnt，用于将系统时钟分频以满足指定波特率。在每个波特周期计数完成或发送触发信号到来时清零计数器，否则每个时钟上升沿累加 1，实现对波特周期的精确控制。

在代码的72-82行，tx_bit_cnt 用于记录当前发送的数据位编号。在发送使能为低时计数器清零；在 SEND_DATA 状态下，每完成一个波特周期累加 1，发送完 8 位数据后清零，为发送停止位做好准备。

状态机逻辑在代码的89-134行实现。状态机共包含五个状态：IDLE（空闲）、SEND_START（发送起始位）、SEND_DATA（发送数据位）、SEND_STOP（发送停止位）和 SEND_END（发送结束）。状态跳转逻辑通过组合逻辑 tx_state_n 计算下一状态：在空闲状态检测到发送触发信号时进入起始位状态，发送一个波特周期的低电平后进入数据发送状态，依次发送 8 位数据后进入停止位状态，发送一个波特周期的高电平后进入发送结束状态，最终回到空闲状态。状态更新在每个时钟上升沿通过 tx_state <= tx_state_n 完成。

在代码的135-168行是状态机的输出逻辑。在发送使能有效时，根据当前状态控制 uart_tx 输出电平：空闲和结束状态输出高电平，起始位输出低电平，数据发送状态按 tx_bit_cnt 输出对应的数据位，停止位输出高电平；当发送使能无效时，输出保持高电平，保证串口线在空闲时为逻辑 1。

9.3.2.串口接收模块设计

串口接收模块是发射模块的逆过程，设计思路区别不大，但是有如下几点需要注意：

1.接收开始信号，当rx下降沿到来后保持几个时钟周期的低电平，表明进入接收start；

2.接收数据提取位置，前面讲发射的时候都是在波特周期开始的位置变更数据，接收数据提取时需要在rx稳定时刻取数，去波特周期的中间位置取数；

3.最终输出数据锁存，在最后1bit存入寄存器后需要对接收数据锁存，并在之后需要给出数据使能信号，表示输出数据有效；

Module设计如下：

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.
4.  module uart_rx # (
5.      parameter                BPS_NUM      =      16'd434
6.  //  设置波特率为4800时，    bit位宽时钟周期个数:50MHz set 10417   40MHz set 8333
7.  //  设置波特率为9600时，    bit位宽时钟周期个数:50MHz set 5208    40MHz set 4167
8.  //  设置波特率为115200时，bit位宽时钟周期个数:50MHz set 434      40MHz set 347
9.  )
10. (
11.     //input ports
12.     input                    clk,
13.     input                    uart_rx,
14.
15.     //output ports
16.     output reg [7:0]   rx_data,
17.     output reg             rx_en,
18.     output                   rx_finish
19. );
20.
21.     // uart rx state machine's state
22.     localparam   IDLE             = 4'h0;    //空闲状态，等待开始信号到来.
23.     localparam   RECEIV_START = 4'h1;   //接收Uart开始信号，低电平一个波特周期.
24. //接收Uart传输数据信号，此工程定义传输8bit，每个波特周期中间位置取值，8个周期后跳转到stop状态
```

```
25.     localparam   RECEIV_DATA   = 4'h2;      .
26.  //停止状态数据线是高电平，与空闲状态是一致的按照协议标准需要等待一个停止位周期再做状态跳转.
27.     localparam   RECEIV_STOP   = 4'h3;
28.     localparam   RECEIV_END    = 4'h4;      //结束中转状态.
29.
30.     //=============================================================
31.     //wire and reg in the module
32.     //=============================================================
33.     reg     [2:0]          rx_state=0;          //current state of tx state machine当前状态
34.     reg     [2:0]          rx_state_n=0;        //next state of tx state machine     下一个状态
35.     reg     [7:0]          rx_data_reg;         //                接收数据缓冲寄存器
36.     reg                     uart_rx_1d;          //save uart_rx one cycle.        保存uart_rx一个时钟周期
37.     reg                     uart_rx_2d;          //save uart_rx one cycle.       保存uart_rx  前两个时钟周期
38.     wire                    start;               //active when start a byte receive.     检测到start信号标志
39.     reg     [15:0]         clk_div_cnt;         //count for division the clock.         波特周期计数器
40.
41.     //=============================================================
42.     //logic
43.     //=============================================================
44.
45.     //some control single.
46.     always @ (posedge clk)
47.     begin
48.          uart_rx_1d <= `UD uart_rx;
49.          uart_rx_2d <= `UD   uart_rx_1d;
50.     end
51.
52.     assign start       = (!uart_rx) && (uart_rx_1d || uart_rx_2d);
53.     assign rx_finish = (rx_state == RECEIV_END);
54.

55.     //division the clock to satisfy baud rate.波特周期计数器
56.     always @ (posedge clk)
57.     begin
58.          if(rx_state == IDLE || clk_div_cnt == BPS_NUM)
59.               clk_div_cnt     <= `UD 16'h0;
60.          else
61.               clk_div_cnt     <= `UD clk_div_cnt + 16'h1;
62.     end
63.
64.     // receive bit data numbers
65.     //在接收数据状态中，接收的bit位计数，每一个波特周期计数加1
66.     reg     [2:0]          rx_bit_cnt=0;       //the bits number has transmited.
67.     always @ (posedge clk)
68.     begin
69.          if(rx_state == IDLE)
70.               rx_bit_cnt <= `UD 3'h0;
71.          else if((rx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
72.               rx_bit_cnt <= `UD 3'h0;
73.          else if((rx_state == RECEIV_DATA) && (clk_div_cnt == BPS_NUM))
74.               rx_bit_cnt <= `UD rx_bit_cnt + 3'h1;
75.          else
76.               rx_bit_cnt <= `UD rx_bit_cnt;
77.     end
78.
```

```
79.  //====================================================================
80.  //receive state machine
81.  //====================================================================
82.      //状态机状态跳转
83.      always @(posedge clk)
84.      begin
85.          rx_state <= rx_state_n;
86.      end
87.
88.      //状态机状态跳转条件及跳转规律
89.      always @ (*)
90.      begin
91.        case(rx_state)
92.            IDLE          :
93.            begin
94.                if(start)                       //监测到start信号到来，下一状态跳转到start状态
95.                    rx_state_n = RECEIV_START;
96.                else
97.                    rx_state_n = rx_state;
98.            end
99.            RECEIV_START     :
100.           begin
101.               if(clk_div_cnt == BPS_NUM)                 //已完成接收start标志信号
102.                   rx_state_n = RECEIV_DATA;
103.               else
104.                   rx_state_n = rx_state;
105.           end
106.           RECEIV_DATA      :
107.           begin
108.               if(rx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM) //已完成8bit数据的传输
109.                   rx_state_n = RECEIV_STOP;
110.               else
111.                   rx_state_n = rx_state;
112.           end
113.           RECEIV_STOP      :
114.           begin
115.               if(clk_div_cnt == BPS_NUM)                         //已完成接收stop标志信号
116.                   rx_state_n = RECEIV_END;
117.               else
118.                   rx_state_n = rx_state;
119.           end
120.           RECEIV_END       :
121.           begin
122.               if(!uart_rx_1d)   //数据线重新被拉低，新数据传输又发送start信号，需要跳转到start状态
123.                   rx_state_n = RECEIV_START;
124.               else                  //没有其他状况出现时，回到空闲状态，等待start信号的到来
125.                   rx_state_n = IDLE;
126.           end
127.           default      :   rx_state_n = IDLE;
128.       endcase
129.     end
130.
131.     // 状态机输出
132.     always @ (posedge clk)
133.     begin
```

```
134.                case(rx_state)
135.                    IDLE            ,
136.                    RECEIV_START :   //在空闲和start状态时将接收数据缓冲寄存器和数据使能置位；
137.                    begin
138.                        rx_en <= `UD 1'b0;
139.                        rx_data_reg <= `UD 8'h0;
140.                    end
141.                    RECEIV_DATA   :
142.                    begin
143.                        if(clk_div_cnt == BPS_NUM[15:1])      //在波特周期的中间位置取传输的数据；
144.        //以循环右移的方式将uart_rx数据填入缓冲寄存器的最高位（Uart传输低位在前，最后一个bit刚好是最高位）
145.                            rx_data_reg   <= `UD {uart_rx , rx_data_reg[7:1]};
146.                    end
147.                    RECEIV_STOP   :
148.                    begin
149.                        rx_en     <= `UD 1'b1;                // 输出使能信号，表示最新的数据输出有效
150.                        rx_data <= `UD rx_data_reg;          // 将缓冲寄存器的值赋值给输出寄存器
151.                    end
152.                    RECEIV_END      :
153.                    begin
154.                        rx_data_reg <= `UD 8'h0;
155.                    end
156.                    default:      rx_en <= `UD 1'b0;
157.                endcase
158.            end
159.
160.        endmodule
```

串口接收模块代码的46行到158行是 UART 串口接收控制逻辑，主要用于从串口输入 uart_rx 接收字节数据，并输出接收数据 rx_data 与有效信号 rx_en，同时通过 rx_finish 指示接收过程是否完成。

46-50行是 uart_rx 信号同步逻辑，通过寄存器 uart_rx_1d 和 uart_rx_2d 在每个时钟上升沿采样 uart_rx 信号，消除异步信号干扰，并用于检测起始位。紧接着在52行，通过组合逻辑 start 检测字节起始信号，即串口线从高电平下降到低电平时标记一个字节接收开始。rx_finish 信号在状态机进入 RECEIV_END 状态时拉高，表示接收过程完成。

而56-62行是波特率周期计数器 clk_div_cnt，用于将系统时钟分频以满足指定波特率。在空闲状态或计数达到波特周期时计数器清零，否则每个时钟上升沿累加 1。

在代码的67-77行，rx_bit_cnt用于记录当前接收的数据位编号。在空闲状态清零计数器；在RECEIV_DATA状态下，每完成一个波特周期累加1，接收完8位数据后清零，为接收停止位做好准备。

状态机逻辑在代码的83-129行实现。状态机共包含五个状态：IDLE（空闲）、RECEIV_START（接收起始位）、RECEIV_DATA（接收数据位）、RECEIV_STOP（接收停止位）和 RECEIV_END（接收结束）。状态跳转逻辑通过组合逻辑rx_state_n计算下一状态：空闲状态检测到start信号时进入起始位状态，接收一个波特周期的低电平后进入数据接收状态，依次接收8位数据后进入停止位状态，接收一个波特周期的高电平后进入结束状态，最终

回到空闲状态；如果在结束状态检测到串口线被拉低，表示新字节开始接收，则跳转到起始位状态。状态更新在每个时钟上升沿通过rx_state <= rx_state_n完成。

132-158行是状态机的输出逻辑。在空闲和起始位状态，将接收数据缓冲寄存器和数据使能清零；在数据接收状态，通过在波特周期中间采样串口线的电平，将数据按低位先行的顺序填入缓冲寄存器；在停止位状态，将缓冲寄存器的值赋给输出寄存器rx_data并拉高rx_en表示数据有效；在结束状态清零缓冲寄存器，为下一次接收做好准备。

9.3.3.串口发送控制模块设计

目标：产生1S间隔的触发信号并输出第一个发送字节，busy的下降沿时输出下一个字节；

Module如下：

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module uart_data_gen(
4.      input                   clk,
5.      input       [7:0]       read_data,
6.      input                   tx_busy,
7.      input       [7:0]       write_max_num,
8.      output reg [7:0]        write_data,
9.      output reg              write_en
10.   );
11.
12.         // set every second send a string,"====HELLO WORLD==="
13.         //  设置约每秒发送一个字符串
14.         reg [25:0] time_cnt=0;
15.         reg [ 7:0] data_num;
16.         always @(posedge clk)
17.         begin
18.             time_cnt <= `UD time_cnt + 26'd1;
19.         end
20.
21.         //  设置串口发射工作区间
22.         reg             work_en=0;
23.         reg             work_en_1d=0;
24.         always @(posedge clk)
25.         begin
26.             if(time_cnt == 26'd2048)
27.                 work_en <= `UD 1'b1;
28.             else if(data_num == write_max_num-1'b1)
29.                 work_en <= `UD 1'b0;
30.         end
31.
32.         always @(posedge clk)
33.         begin
34.             work_en_1d <= `UD work_en;
35.         end
36.
37.         // get the tx_busy's falling edge     获取tx_busy的下降沿
38.         reg                 tx_busy_reg=0;
39.         wire                tx_busy_f;
```

```
40.         always @ (posedge clk) tx_busy_reg <= `UD tx_busy;
41.
42.         assign tx_busy_f = (!tx_busy) && (tx_busy_reg);
43.
44.         // 串口发射数据触发信号
45.         reg write_pluse;
46.         always @ (posedge clk)
47.         begin
48.             if(work_en)
49.             begin
50.                 if(~work_en_1d || tx_busy_f)
51.                     write_pluse <= `UD 1'b1;
52.                 else
53.                     write_pluse <= `UD 1'b0;
54.             end
55.             else
56.                 write_pluse <= `UD 1'b0;
57.         end
58.
59.         always @ (posedge clk)
60.         begin
61.             if(~work_en & tx_busy_f)
62.                 data_num     <= 7'h0;
63.             else if(write_pluse)
64.                 data_num     <= data_num + 8'h1;
65.         end
66.
67.         always @(posedge clk)
68.         begin
69.             write_en <= `UD write_pluse;
70.         end
71.
72.     //  字符的对应ASCII码
73.         always @ (posedge clk)
74.         begin
75.             case(data_num)
76.                 8'h0  ,
77.                 8'h1  : write_data <= `UD 8'h77;// ASCII code is w
78.                 8'h2  : write_data <= `UD 8'h77;// ASCII code is w
79.                 8'h3  : write_data <= `UD 8'h77;// ASCII code is w
80.                 8'h4  : write_data <= `UD 8'h2E;// ASCII code is .
81.                 8'h5  : write_data <= `UD 8'h6D;// ASCII code is m
82.                 8'h6  : write_data <= `UD 8'h65;// ASCII code is e
83.                 8'h7  : write_data <= `UD 8'h79;// ASCII code is y
84.                 8'h8  : write_data <= `UD 8'h65;// ASCII code is e
85.                 8'h9  : write_data <= `UD 8'h73;// ASCII code is s
86.                 8'ha  : write_data <= `UD 8'h65;// ASCII code is e
87.                 8'hb  : write_data <= `UD 8'h6D;// ASCII code is m
88.                 8'hc  : write_data <= `UD 8'h69;// ASCII code is i
89.                 8'hd  : write_data <= `UD 8'h2E;// ASCII code is .
90.                 8'he  : write_data <= `UD 8'h63;// ASCII code is c
91.                 8'hf  : write_data <= `UD 8'h6F;// ASCII code is o
92.                 8'h10 : write_data <= `UD 8'h6D;// ASCII code is m
93.                 8'h11 ,
94.                 8'h12 : write_data <= `UD 8'h0d;
```

```
95.                     8'h13 : write_data <= `UD 8'h0a;
96.                     default : write_data <= `UD read_data;
97.                 endcase
98.             end
99.
100. .     endmodule
```

串口发送控制模块，用于按照固定时间间隔生成指定字符序列并通过串口发送。模块以系统时钟 clk 为基础，通过控制逻辑管理发送触发和数据输出。

代码的14行到98行是串口数据生成与发送控制逻辑，主要功能是每隔一定时间生成一个指定字符串（如 "www.meyesemi.com\r\n"）并通过 write_data 和 write_en 输出给 UART 发送模块。

首先，代码的14-19行是时间计数器time_cnt，每个时钟上升沿累加1，用于实现发送周期控制，本实验中是实现功能是每秒触发一次字符串发送。

在22-30行，通过 work_en 信号控制串口发送工作区间：当time_cnt达到设定值时开始发送，当发送的数据量达到write_max_num-1时结束发送。work_en_1d在32-35行用于寄存器同步，实现发送使能的边沿检测。

38-42行通过寄存器tx_busy_reg同步tx_busy信号，并生成tx_busy_f信号，用于检测UART发送模块的忙状态下降沿，作为发送触发条件之一。

代码的45-57行生成write_pluse发送脉冲信号：在工作使能状态下，如果刚开始发送或检测到tx_busy下降沿，则拉高write_pluse，触发 UART 发送模块发送一个字节。

在59-65行，data_num 用于记录当前发送的字节序号：当发送结束且tx_busy下降沿出现时清零；当write_pluse有效时累加，为依次输出字符做索引。write_en在67-70行直接由write_pluse 同步输出。

在73-98行，通过case语句将data_num对应到具体ASCII字符，实现字符映射输出：如索引0-2输出'w'，索引4输出'.'，索引5-10输出"meyese"等，最后两个索引输出回车换行0x0d 0x0a，default情况下输出read_data。

9.3.4.串口实验顶层模块设计

目标：板子1s向串口助手发送一次十进制显示的“www.meyesemi.com”，通过串口助手向板子以十六进制形式发送数字，LED以二进制显示亮起。

Uart_data_gen模块产生一个间隔1S钟的触发信号，同时输出第一个发送字节，等待uart_tx输出的busy下降沿到来，获知uart_tx进入空闲状态可发送下一个byte时，再次给出串口发送的触发脉冲，并输出下一个字节；

Uart_rx模块接收到数据后输出一个rx_en信号（接收数据使能信号）、一组接收数据信号；接收的数据信号是锁存的，可直接点亮LED灯；

具体的module实现如下：

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.
4.  module uart_top(
5.      //input ports
6.      input           clk,
7.      input           uart_rx,
8.
9.      //output ports
10.     output  [7:0] led,
11.     output          uart_tx);
12. );
13.
14.    parameter        BPS_NUM = 16'd434;
15.    // 设置波特率为4800时，bit位宽时钟周期个数:50MHz set 10417   40MHz set 8333
16.    // 设置波特率为9600时，bit位宽时钟周期个数:50MHz set 5208    40MHz set 4167
17.    // 设置波特率为115200时，bit位宽时钟周期个数:50MHz set 434   40MHz set 347 12M set 104
18.
19.
20. //====================
21. //wire and reg in the module
22. //====================
23.
24.     wire            tx_busy;     //transmitter is free.
25.     wire            rx_finish;   //receiver is free.
26.     wire    [7:0] rx_data;       //the data receive from uart_rx.
27.     wire    [7:0] tx_data;
28.     wire            tx_en;       //enable transmit.
29.
30. //=======================
31. //logic
32. //=======================
33.     wire rx_en;
34. //=======================
35. //instance
36. //=======================
37.     reg   [7:0] receive_data;
38.     always @(posedge clk)   receive_data <= led;
39.     uart_data_gen uart_data_gen(
40.         .clk                (clk          ),    //input           clk,
41.         .read_data          (receive_data ),    //input   [7:0]   read_data,
42.         .tx_busy            (tx_busy      ),    //input           tx_busy,
43.         .write_max_num  (8'h14            ),    //input   [7:0]   write_max_num,
44.         .write_data         (tx_data      ),    //output reg [7:0]  write_data
45.         .write_en           (tx_en        )     //output reg        write_en
46.     );
47.
48.     //uart transmit data module.
49.     uart_tx #(
50.          .BPS_NUM        (  BPS_NUM     ) //parameter BPS_NUM   =     16'd434
51.      )
52.      u_uart_tx(
53.         .clk                (  clk           ),// input           clk,
```

```
54.             .tx_data                ( tx_data           ),// input [7:0]         tx_data,
55.             .tx_pluse               ( tx_en             ),// input                tx_pluse,
56.             .uart_tx                ( uart_tx           ),// output reg           uart_tx,
57.             .tx_busy                ( tx_busy           ) // output               tx_busy
58.         );
59.
60.         //Uart receive data module.
61.         uart_rx #(
62.                 .BPS_NUM            ( BPS_NUM           ) //parameter  BPS_NUM  =     16'd434
63.          )
64.          u_uart_rx (
65.             .clk                    ( clk               ),//
    input                   clk,
66.             .uart_rx                ( uart_rx           ),// input                uart_rx,
67.             .rx_data                ( rx_data           ),// output reg [7:0]     rx_data,
68.             .rx_en                  ( rx_en             ),// output reg           rx_en,
69.             .rx_finish              ( rx_finish         ) // output               rx_finish
70.         );
71.
72.         assign led = rx_data;
73.
74.  endmodule
```

模块进行信号声明与内部连线数据生成模块实例化，实例化 uart_data_gen 模块，用于定时生成串口发送数据。UART 发送模块实例化（uart_tx），负责将 tx_data 按波特率发送到 uart_tx 输出端。UART 接收模块实例化（uart_rx）接收外部串口信号 uart_rx，并将接收到的数据输出到 rx_data，同时生成 rx_en 和 rx_finish 信号。37-38 行使用寄存器 receive_data 缓存 LED 状态。72 行将 rx_data 直接输出到 LED，实现接收到的数据通过 LED 显示。

顶层模块主要是整合 UART 接收、数据生成和 UART 发送模块，实现了串口数据收发与 LED 显示功能。接收到的数据实时显示在 LED 上，同时模块可按照设定周期生成数据并通过 UART 发送，实现串口双向通信。

9.4. 实验现象

用SSCOM串口调试工具，波特率设置为115200bps，数据格式为1位起始位、8位数据位、无校验位、1位结束位，用Type-C连接开发板与电脑后有如下现象：

实验现象一：在串口工具中每隔1S中打印一次：“www.meyesemi.com”；

实验现象二：

在串口工具上以Hex格式发送55；我们可看到OPHW-25板卡上的LED0,LED2为熄灭，LED1,LED3为点亮状态；

发送F0；我们可看到OPHW-25板卡上的LED0~3为熄灭，。

10.HDMI 实验例程

10. 1. 实验简介

实验目的：

OPHW-25开发板通过HDMI在屏幕上显示彩条；

实验环境：

Window11

PDS2022.2

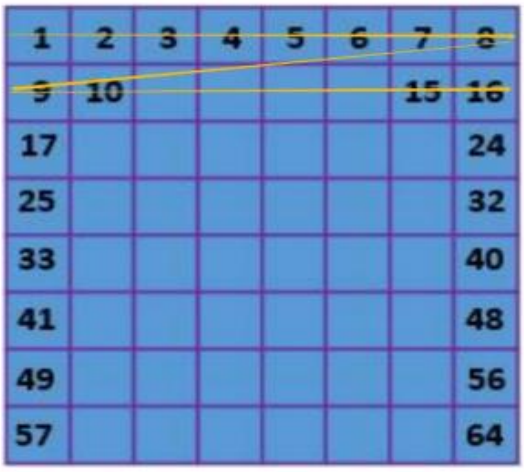
硬件环境：

OPHW-25开发板

10. 2. 实验原理

10.2.1.显示原理

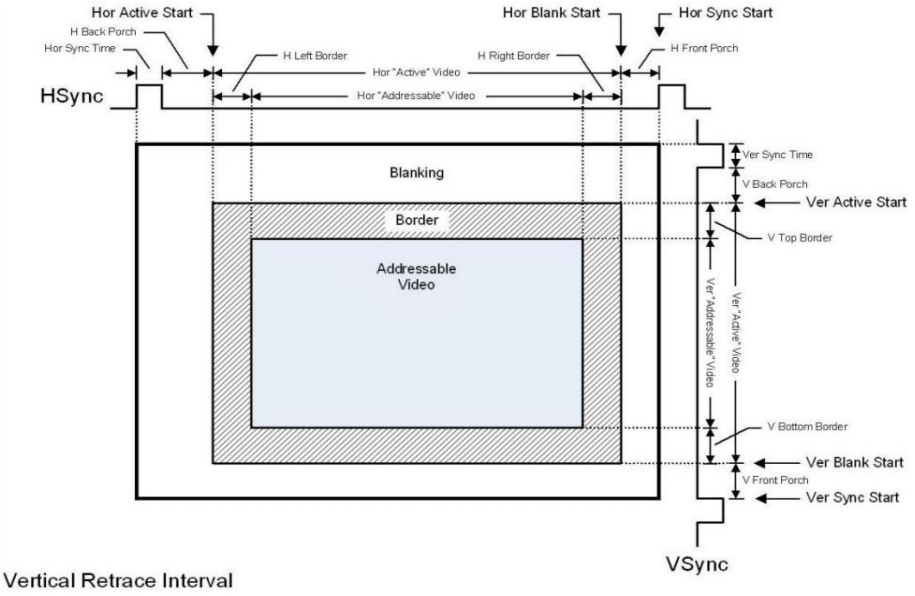
下图表示一个8*8像素的画面，图中每个格子表示一个像素点，显示图像时像素点快速点亮的过程按表格中编号的顺序逐个点亮，从左到右，从上到下，按图中箭头方向的“Z”字形顺序。



以上图为例，每行8个像素点，每完成一行信号的传输，会转到下一行信号传输，直到完成第8行数据的传输，就完成了一个画面的数据传输了，一个画面也称为一场或一帧，显示每秒中刷新的帧数称为帧率。比如1920*1080P像素，就是1行有效像素点1920，一场（也就是一帧）有效行为1080行。

每个像素点的像素值数据，对应每个像素点的颜色。常见的像素值表示格式比如：RGB888，RGB分别代表：红R,绿G，蓝B，888是指R、G、B分别有8bit，也就是R、G、B每一色光有2^8=256

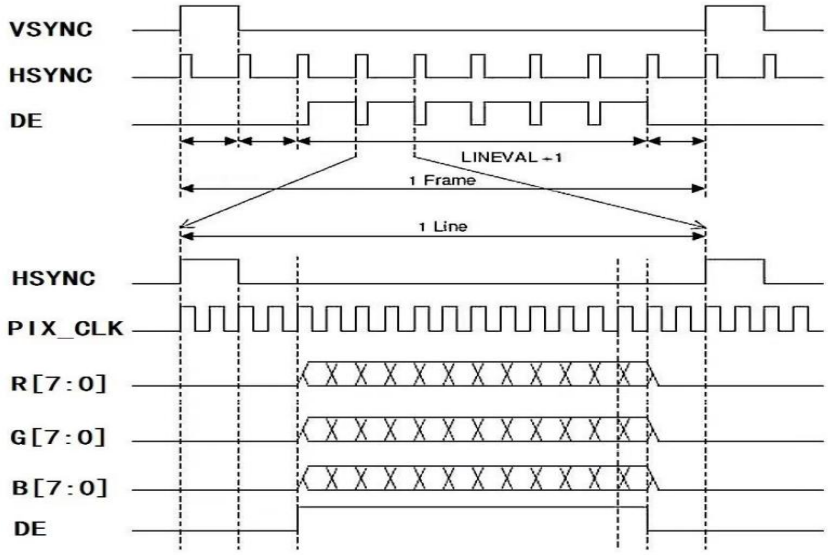
级阶调，通过RGB三色光的不同组合，一个像素上最多可显示24位的256*256*256=16,777,216色。



像素数据源源不断输送进来，行、场的切换通过行场同步信号来控制，即hsync（行同步）和vsync（场同步信号）。

上图中Addressable部分内容是在显示器中可看到的区域，像素点是否有效通过DE信号标识；Border可理解为显示黑边或者显示边框，通常Border显示的像素值是0（黑色）。行、场切换过程都是在用户感受不到的区域进行的，这个区域就是Blanking部分，称为消隐区间。同步信号上升沿表示新的一行/一场开始，Hsync对应行，Vsync对应场。

彩条产生：



本实验采用1920*1080@60的视频规格，详细时序参数如下：

VESA MONITOR TIMING STANDARD

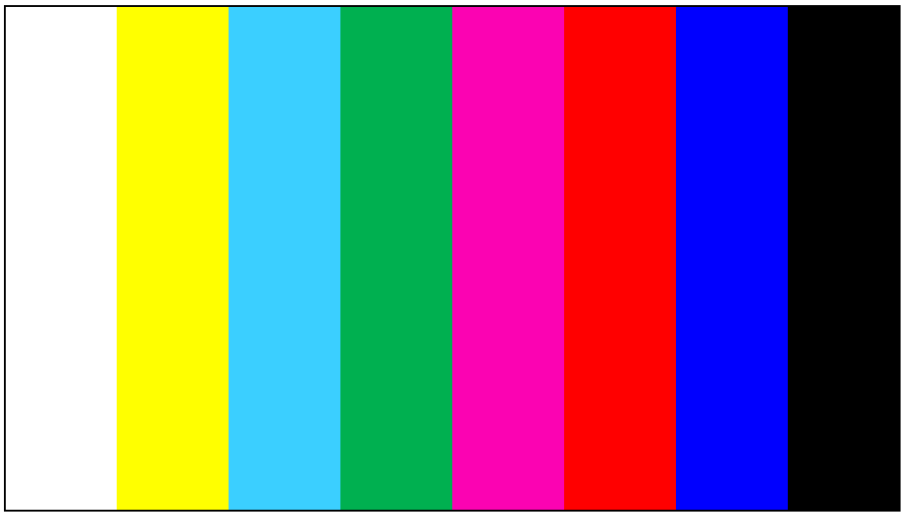
Adopted: 11/17/08
Resolution: 1920 x 1080 at 60 Hz (non-interlaced)
EDID ID: DMT ID: 52h; STD 2 Byte Code: (D1, C0)h; CVT 3 Byte Code: n/a
Method: *** NOT CVT COMPLIANT ***
Per CEA-861 --- 1080p (Code 16) Timing Definition

Detailed Timing Parameters

Timing Name	= **1920 x 1080 @ 60Hz;**						
Hor Pixels	= **1920;**	// Pixels					
Ver Pixels	= **1080;**	// Lines					
Hor Frequency	= 67.500;	// kHz	=	14.8 usec	/	line	
Ver Frequency	= 60.000;	// Hz	=	16.7 msec	/	frame	
Pixel Clock	= **148.500;**	// MHz	=	6.7 nsec	± 0.5%		
Character Width	= **4;**	// Pixels	=	26.9 nsec			
Scan Type	= **NONINTERLACED;**			// H Phase	=	1.4 %	
Hor Sync Polarity	= **POSITIVE**	// HBlank	=	12.7% of HTotal			
Ver Sync Polarity	= **POSITIVE**	// VBlank	=	4.0% of VTotal			
Hor Total Time	= 14.815;	// (usec)	=	550 chars	=	2200 Pixels	
Hor Addr Time	= 12.929;	// (usec)	=	480 chars	=	1920 Pixels	
Hor Blank Start	= 12.929;	// (usec)	=	480 chars	=	1920 Pixels	
Hor Blank Time	= 1.886;	// (usec)	=	70 chars	=	280 Pixels	
Hor Sync Start	= 13.522;	// (usec)	=	502 chars	=	2008 Pixels	
// H Right Border	= 0.000;	// (usec)	=	**0** chars	=	0 Pixels	
// H Front Porch	= 0.593;	// (usec)	=	**22** chars	=	88 Pixels	
Hor Sync Time	= 0.296;	// (usec)	=	**11** chars	=	44 Pixels	
// H Back Porch	= 0.997;	// (usec)	=	**37** chars	=	148 Pixels	
// H Left Border	= 0.000;	// (usec)	=	**0** chars	=	0 Pixels	
Ver Total Time	= 16.667;	// (msec)	=	1125 lines	HT – (1.06xHA)		
Ver Addr Time	= 16.000;	// (msec)	=	1080 lines	= 1.11		
Ver Blank Start	= 16.000;	// (msec)	=	1080 lines			
Ver Blank Time	= 0.667;	// (msec)	=	45 lines			
Ver Sync Start	= 16.059;	// (msec)	=	1084 lines			
// V Bottom Border	= 0.000;	// (msec)	=	**0** lines			
// V Front Porch	= 0.059;	// (msec)	=	**4** lines			
Ver Sync Time	= 0.074;	// (msec)	=	**5** lines			
// V Back Porch	= 0.533;	// (msec)	=	**36** lines			
// V Top Border	= 0.000;	// (msec)	=	**0** lines			

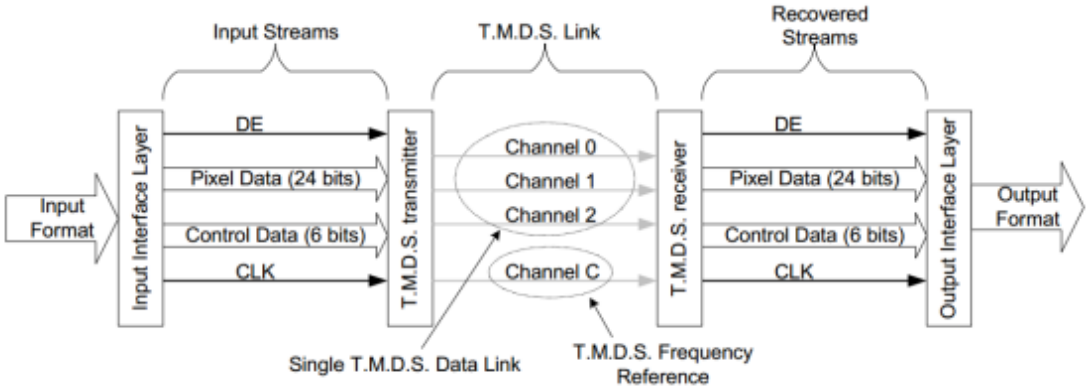
HDMI显示的数据源采用verilog编写的显示时序产生模块sync_vg实现上图的时序，彩条生成模块pattern_vg根据像素点所在位置，即列数和行数确定像素值，实现彩条图案。

彩条按照每行均匀分成8部分，根据每行的像素点数的范围对像素值设置成对应的颜色，实现彩条信号。



10.2.2.HDMI 接口设计

HDMI输出接口采用TMDS通信方式。在一个时钟周期内，每个TMDS通道都能传送10bit的数据流。

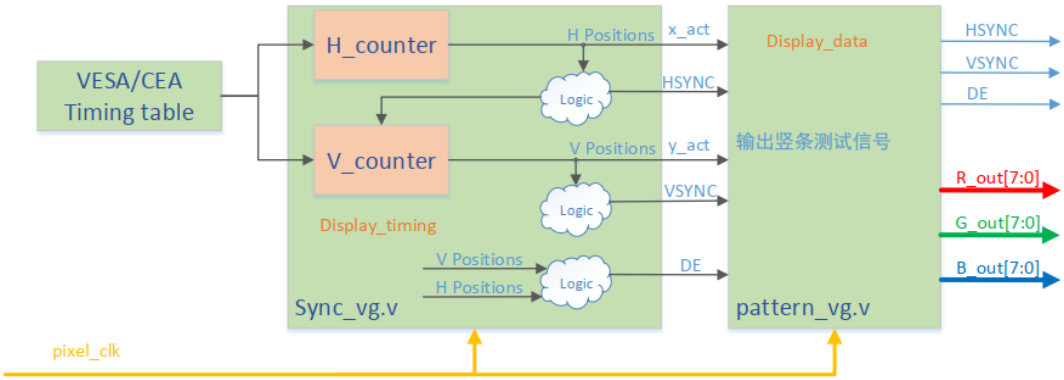


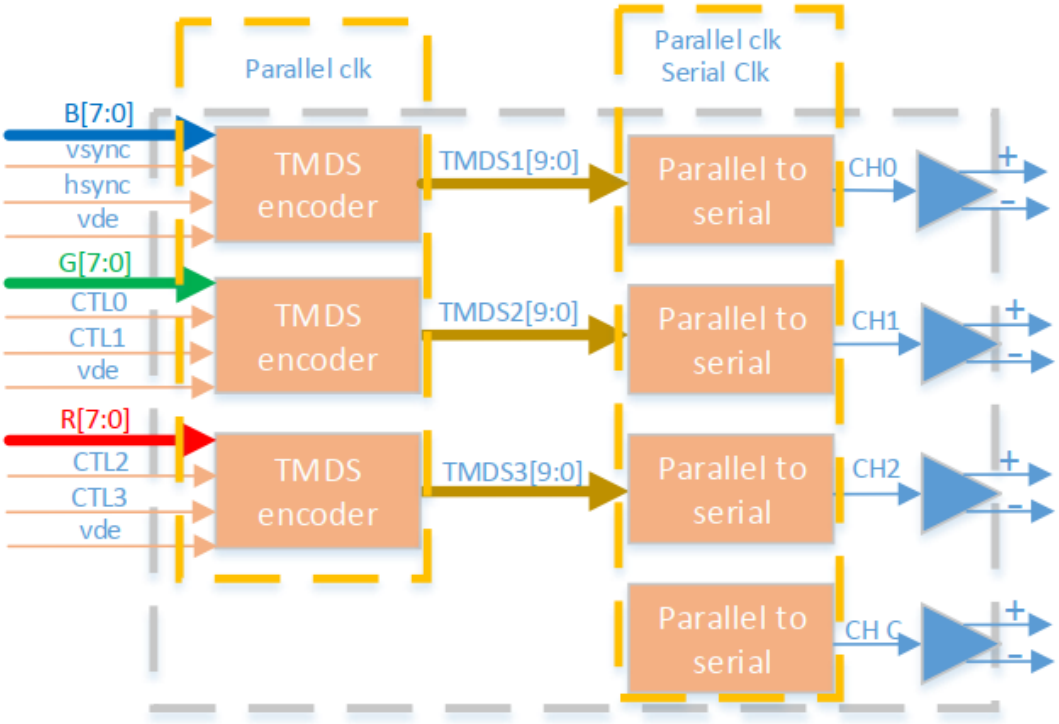
LVDS（Low Voltage Differential Signaling，低电压差分信号）是一种以低摆幅、差分方式进行高速数据传输的接口技术，最初用于高速串行通信，如今在显示接口、板级互连和芯片间通信中被广泛采用。它的设计初衷并不是追求极高的单通道带宽，而是在高速、低功耗和高可靠性之间取得平衡。LVDS 最突出的优势是抗干扰能力强。它采用一对差分信号线进行传输，接收端只关心两根线之间的电压差，而对共模噪声不敏感。当外界电磁干扰同时作用在两根线上时，这部分噪声会被差分接收器自动抵消，因此 LVDS 在复杂电磁环境中依然能够保持较低的误码率，非常适合高速数字系统和密集布线的电路板。

10.2.3.实验源码设计

实验hdmi_test

HDMI输出彩条显示例程，分成4个模块，时钟模块pll、、显示时序产生模块sync_vg、彩条生成模块pattern_vg，编码转换rgb转tmds模块rgb2tmds以下为模块拓扑图，源码详情请查看demo。





10. 3. 实验现象

实验现象：hdmi_test

连接好OPHW-25开发板和显示器，下载程序，可以看到显示器显示8条彩条。

11.DDR3 读写实验例程

11.1. 实验简介

实验目的：

OPHW-25H开发板上配有1颗Micron公司的1Gbit（128MB）的DDR3芯片,型号为MT41K64M16。DDR3的总线宽度共为16bit。DDR3 SDRAM的最高运行时钟速度可达400MHz(数据速率800Mbps)。该DDR3存储系统直接连接到FPGA。实验生成DDR3 IP官方例程，实现DDR3的读写控制，了解其工作原理和用户接口。

实验环境：

Window11

PDS2022.2

硬件环境：

OPHW-25开发板

11.2. DDR3 控制器简介

OPHW-25为用户提供一套完整的DDR memory控制器解决方案，配置方式比较灵活，采用软核实现DDR memory的控制，有如下特点：

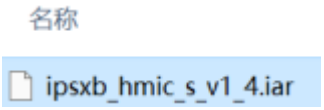
- 支持 DDR3
- 支持 x8、 x16 Memory Device
- 最大位宽支持 32 bit
- 支持裁剪的 AXI4 总线协议
- 一个 AXI4 256 bit Host Port
- 支持 Self_refresh， Power down
- 支持 Bypass DDRC
- 支持 DDR3 Write Leveling 和 DQS Gate Training
- DDR3 最快速率达 800 Mbps

11.3. 实验设计

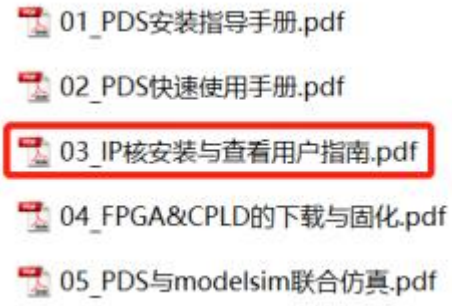
11.3.1.安装 DDR3 IP 核

PDS安装后，需手动添加DDR3IP，请按以下步骤完成：

（1）DDR3IP文件： ipsxb_hmic_s_v1_4.iar

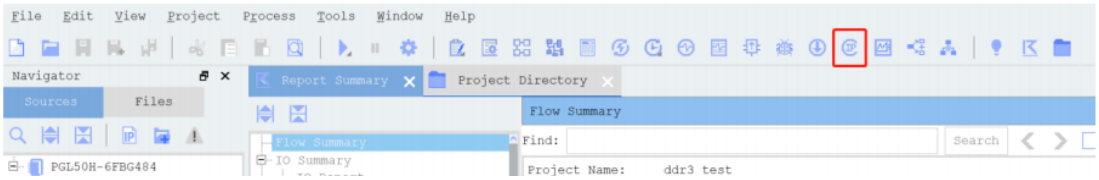


（2）IP安装步骤：IP核安装与查看用户指南.pdf

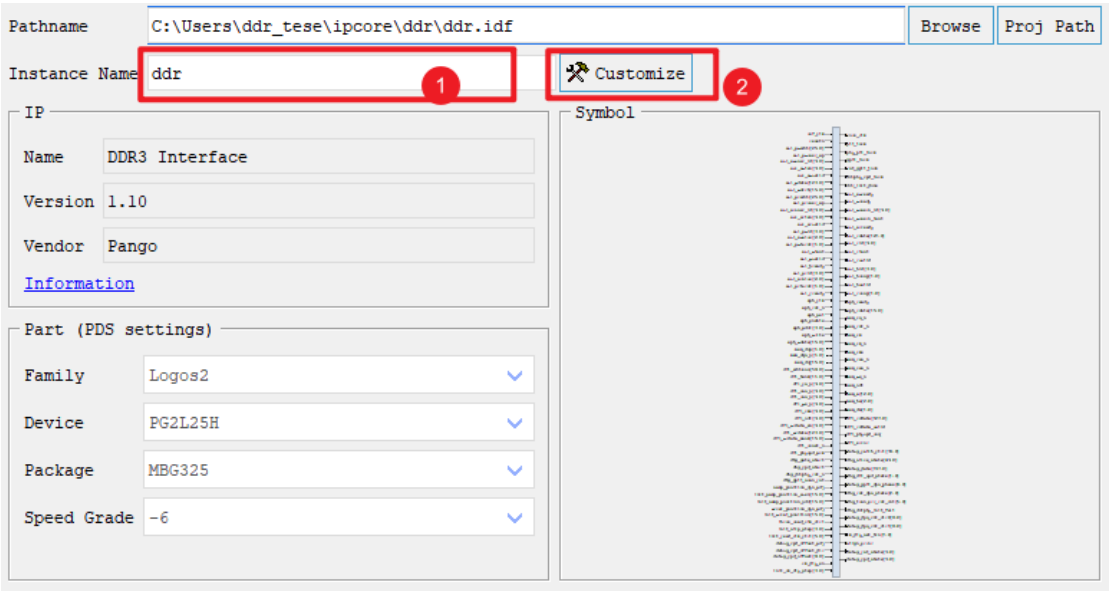


11.3.2.DDR3 读写 Example 工程

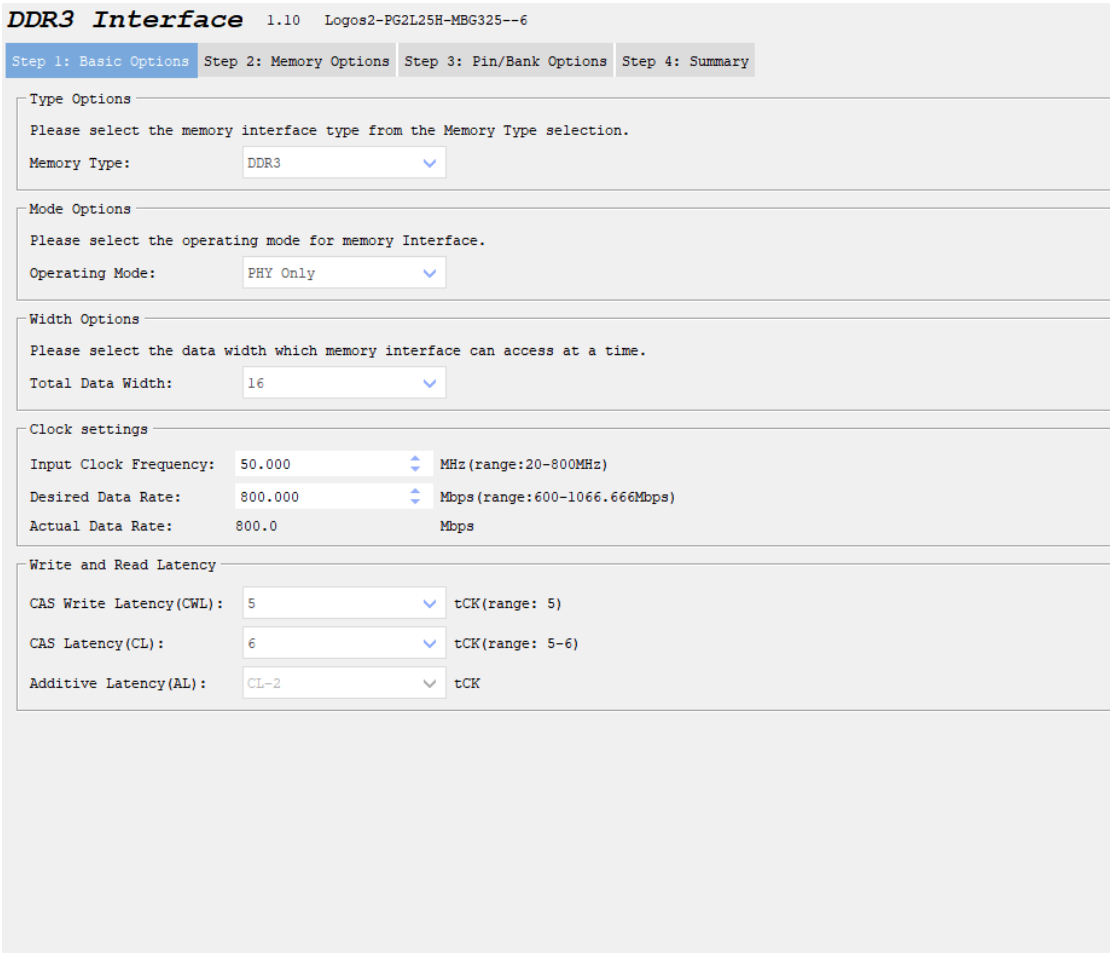
1.打开PDS软件，新建工程ddr_test，点开如下图标，打开IPCompiler；



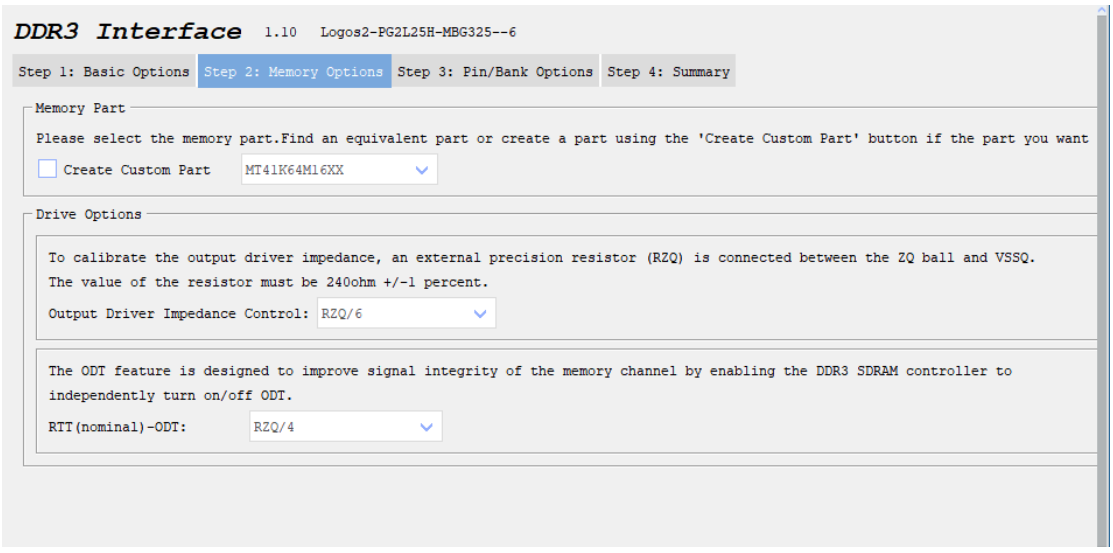
2.选择DDR3IP，取名，然后点击Customize；



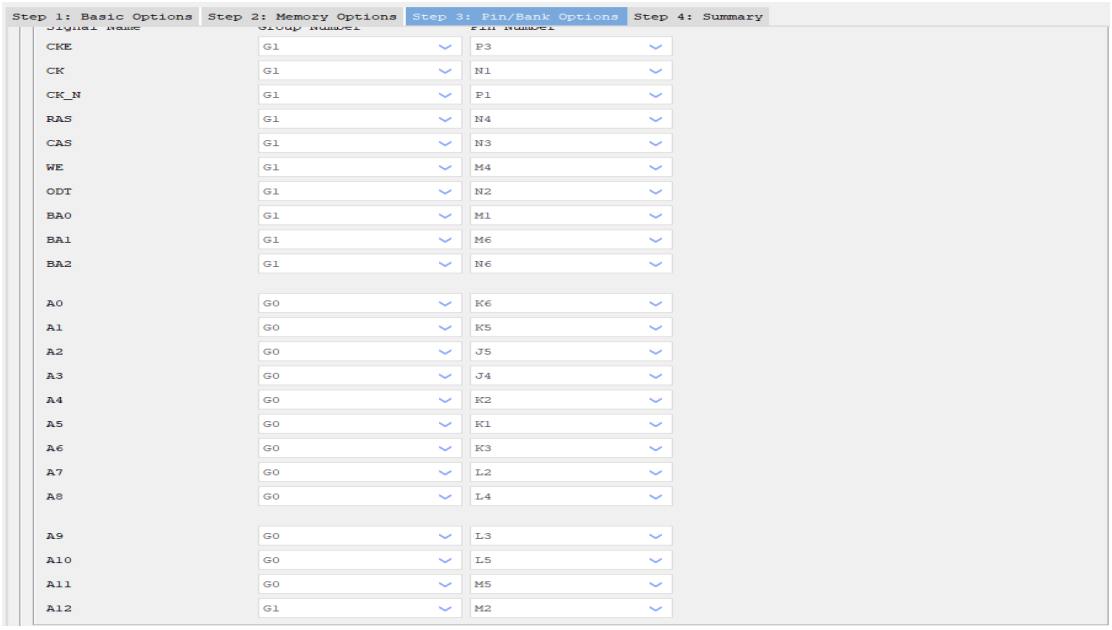
3.在DDR设置界面中Step1按照如下设置：



4.Step2按照如下设置：



5.Step3按照如下设置，勾选CustomControl/AddressGroup，管脚约束参考原理图：



6.Step4为概要，点击Generate可生成DDR3IP；

7.关闭本工程，在本工程路径下打开Example工程：ddr_test\ipcore\ddr_test\pnr

8.打开顶层文件，free_clk、ref_clk可使用同一时钟源：

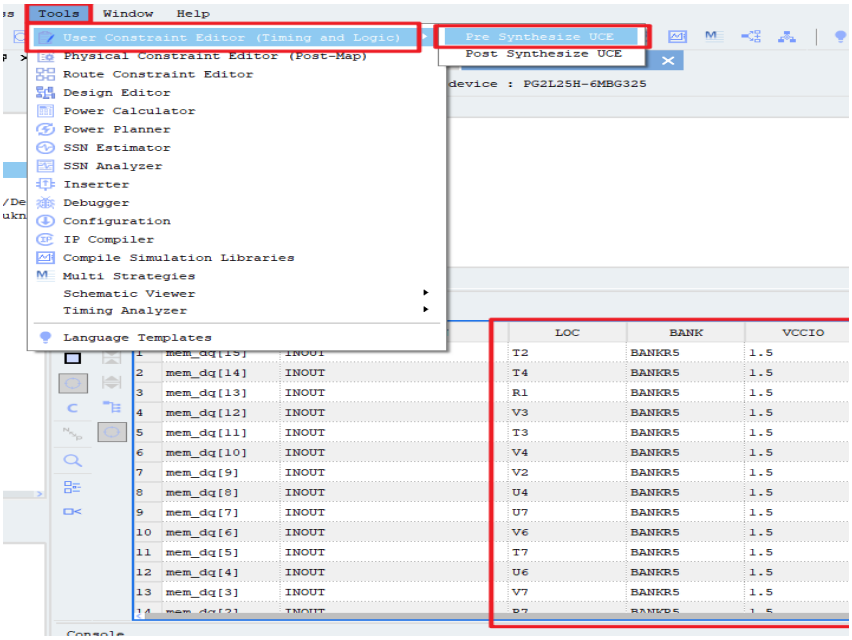
```
33  ) (
34  input                                   ref_clk_p                ,
35  input                                   ref_clk_n                ,
36  //input                                   free_clk                 ,
37  input                                   rst_board                ,
38  output                                  pll_lock                 ,
39  output                                  ddrphy_cpd_lock          ,
40  output                                  ddr_init_done            ,
41  //uart
```

Report Summary | Project Directory | test_ddr.v*

Find: free_clk_g　Search

```
531 wire [6-1:0]                    force_rdvref_value_group        ;
532 wire [8-1:0]                    force_rd_dbi_dly_group          ;
533 wire [64-1:0]                   force_rd_dq_dly_group           ;
534 wire [8-1:0]                    force_dqs_even_dly_group        ;
535 wire [8-1:0]                    force_dqs_odd_dly_group         ;
536 wire [64-1:0]                   force_wr_dq_dly_group           ;
537 wire [8-1:0]                    force_wrdm_dbi_dly_group        ;
538
539 GTP_INBUFDS refclk_inbuf
540 (
541     .O                          (ref_clk                        ),
542     .I                          (ref_clk_p                      ),
543     .IB                         (ref_clk_n                      )
544 );
545
546 assign free_clk = ref_clk;
547
548 GTP_CLKBUFG free_clk_ibufg
549 (
550     .CLKOUT                     (free_clk_g                     ),
551     .CLKIN                      (free_clk                       )
552 );
```

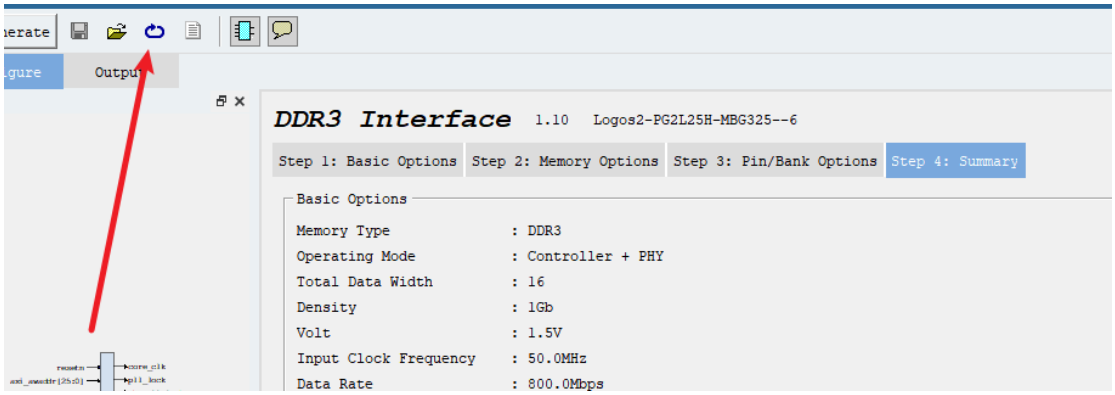
9.对“Step3已做管脚约束”外的其他管脚，对照原理图使用UCE工具进行修改：



10.以下管脚可约束在LED，方便观察实验现象；

信号	LED
err_flag_led	LED0
heart_beat_led	LED1
ddr_init_done	LED2
pll_lock	LED3

11.在第一个工程时创建ddr ip时可按以下方式查看IP核的用户指南，了解Example模块组成；



11. 4. 实验现象

注：例程位置：Demo\ddr_test\ipcore\ddr_test\pnr

信号名称	参考说明	LED 编号
ddr_init_done	初始化标志	1
err_flag_led	数据检测错误信号	0
heart_beat_led	心跳信号	3
ddrphy_cpd_lock	时钟锁定	2

打开约束文件，修改上面的四个信号名的引脚约束，重新约束到开发板上的LED灯。下载程序，可以看到LED0和LED3闪烁，LED1，LED2熄灭。

12.基于 UDP 的以太网传输实验例程

12. 1. 实验简介

实验目的：

完成基于UDP的以太网通信测试。

实验环境：

Window11

PDS2022.2

硬件环境：

OPHW-25开发板

12. 2. 开发板以太网接口简介

OPHW-25开发板使用 Realtek RTL8211E PHY 实现了一个 10/100/1000 以太网端口，用于网络连接。该器件工作电压为支持 2.5V、3.3V，通过 RGMII 接口连接到 OPHW-25。RJ-45 连接器是 HFJ11-1G01E-L12RL，具有集成的自动缠绕磁性元件，可提高性能，质量和可靠性。

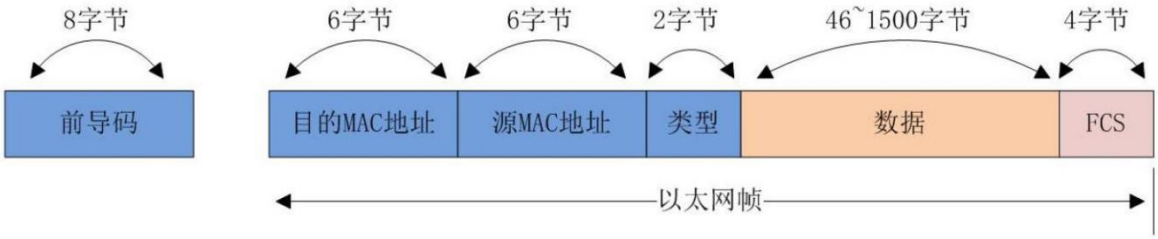
RJ-45 有两个状态指示灯 LED，用于指示流量和有效链路状态（详情请查看“OPHW-25开发板硬件使用手册”）。

12. 3. 实验要求

通过以太网端口实现 PC 端和开发板间通信，实现了 ARP，UDP 功能。

12. 4. 以太网协议简介

12.4.1.以太网帧格式



前导码（Preamble）：8 字节，连续7 个8’h55 加1 个8’hd5，表示一个帧的开始，用于双方设备数据的同步。

目的MAC 地址：6 字节，存放目的设备的物理地址，即MAC 地址；

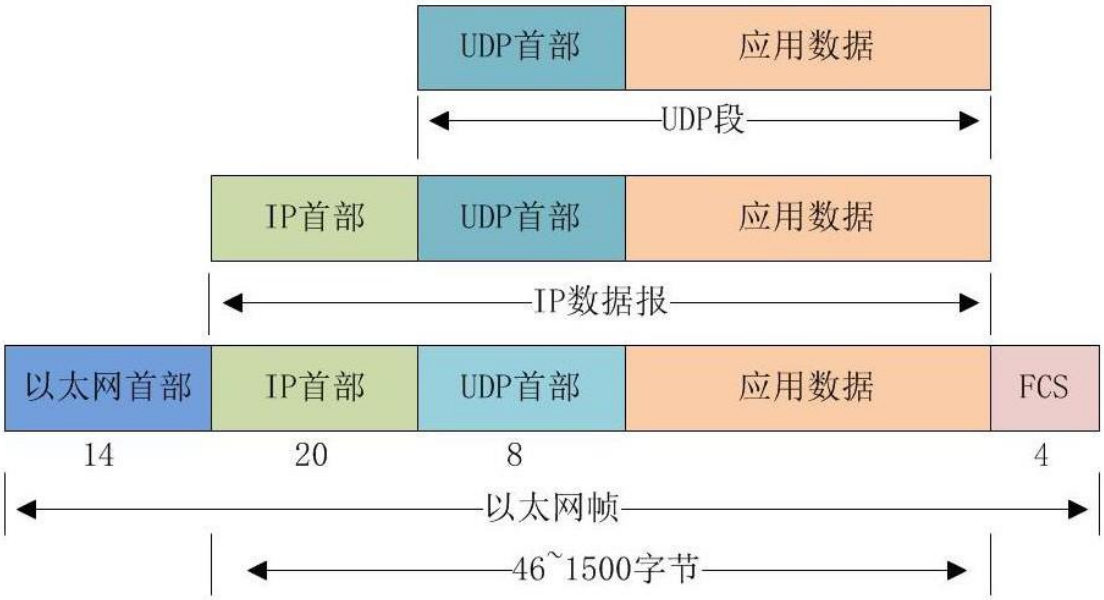
源MAC 地址：6 字节，存放发送端设备的物理地址；

类型：2 字节，用于指定协议类型，常用的有0800 表示IP 协议，0806 表示ARP 协议，8035表示RARP 协议；

数据：46 到1500 字节，最少46 字节，不足需要补全46 字节，例如IP 协议层就包含在数据部分，包括其IP 头及数据。

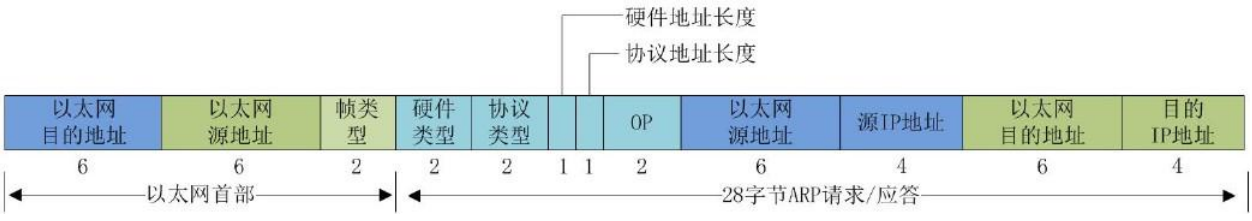
FCS：帧尾，4 字节，称为帧校验序列，采用32 位CRC 校验，对目的MAC 地址字段到数据字段进行校验。

进一步扩展，以 UDP 协议为例，可以看到其结构如下，除了以太网首部的 14 字节，数据部分包含 IP 首部，UDP 首部，应用数据共 46~1500 字节。



12.4.2.ARP 数据报格式

ARP 地址解析协议，即 ARP（Address Resolution Protocol），根据 IP 地址获取物理地址。主机发送包含目的 IP 地址的 ARP 请求广播（MAC 地址为 48'hff_ff_ff_ff_ff_ff）到网络上的主机，并接收返回消息，以此确定目标的物理地址，收到返回消息后将 IP 地址和物理地址保存到缓存中，并保留一段时间，下次请求时直接查询 ARP 缓存以节约资源。下图为 ARP 数据报格式。



帧类型：ARP 帧类型为两字节0806；

硬件类型：指链路层网络类型，1 为以太网；

协议类型：指要转换的地址类型，采用0x0800 IP 类型，之后的硬件地址长度和协议地址长度分别对应6 和4；

OP 字段中 1 表示 ARP 请求，2 表示 ARP 应答

例如：|ff ff ff ff ff ff|00 0a 35 01 fe c0|08 06|00 01|08 00|06|04|00 01|00 0a

35 01 fe c0|c0 a8 00 02| ff ff ff ff ff ff|c0 a8 00 03|

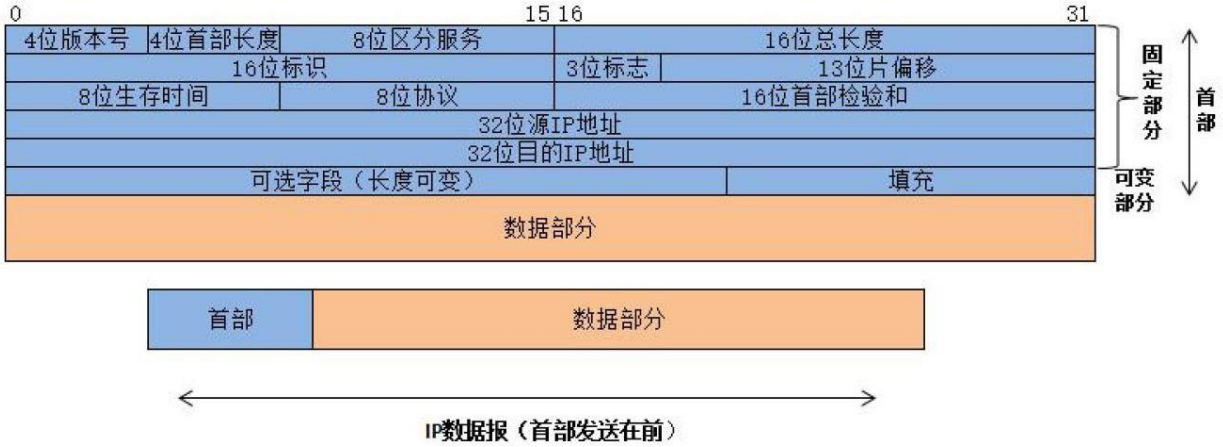
表示向 192.168.0.3 地址发送 ARP 请求。

|00 0a 35 01 fe c0 | 60 ab c1 a2 d5 15 |08 06|00 01|08 00|06|04|00 02| 60 ab c1 a2 d5 15|c0 a8 00 03|00 0a 35 01 fe c0|c0 a8 00 02|

表示向 192.168.0.2 地址发送 ARP 应答。

12.4.3.IP 数据包格式

因为 UDP 协议包只是 IP 包中的一种，所以我们来介绍一下 IP 包的数据格式。下图为IP分组的报文头格式，报文头的前 20 个字节是固定的，后面的可变



版本:占4 位,指IP 协议的版本目前的IP 协议版本号为4 (即IPv4)；

首部长度:占4 位,可表示的最大数值是15 个单位(一个单位为4 字节)因此IP 的首部长度的最大值是60 字节；

区分服务:占8 位,用来获得更好的服务,在旧标准中叫做服务类型,但实际上一直未被使用过1998年这个字段改名为区分服务.只有在使用区分服务(DiffServ)时,这个字段才起作用.一般的情况下都不使用这个字段；

总长度:占16 位,指首部和数据之和的长度,单位为字节,因此数据报的最大长度为65535字节总长度必须不超过最大传送单元MTU

标识:占16 位,它是一个计数器,用来产生数据报的标识

标志(flag):

占 3 位,目前只有前两位有意义

MF：标志字段的最低位是 MF (More Fragment)，MF=1 表示后面“还有分片”。MF=0 表示最后一个分片

DF：标志字段中间的一位是 DF (Don't Fragment)，只有当 DF=0 时才允许分片

片偏移:占12 位,指较长的分组在分片后某片在原分组中的相对位置.片偏移以8 个字节为偏移单位；

生存时间:占8 位,记为TTL (Time To Live) 数据报在网络中可通过的路由器数的最大值,TTL 字段是由发送端初始设置一个8 bit 字段.推荐的初始值由分配数字RFC 指定,当前值为64.发送ICMP 回显应答时经常把TTL 设为最大值255；

协议:占8 位,指出此数据报携带的数据使用何种协议以便目的主机的IP 层将数据部分上交给哪个处理过程, 1 表示为ICMP 协议, 2 表示为IGMP 协议, 6 表示为TCP 协议, 17 表示为UDP 协议；

首部检验和:占16 位,只检验数据报的首部不检验数据部分，采用二进制反码求和，即将16 位数据相加后，再将进位与低16 位相加，直到进位为0，最后将16 位取反；

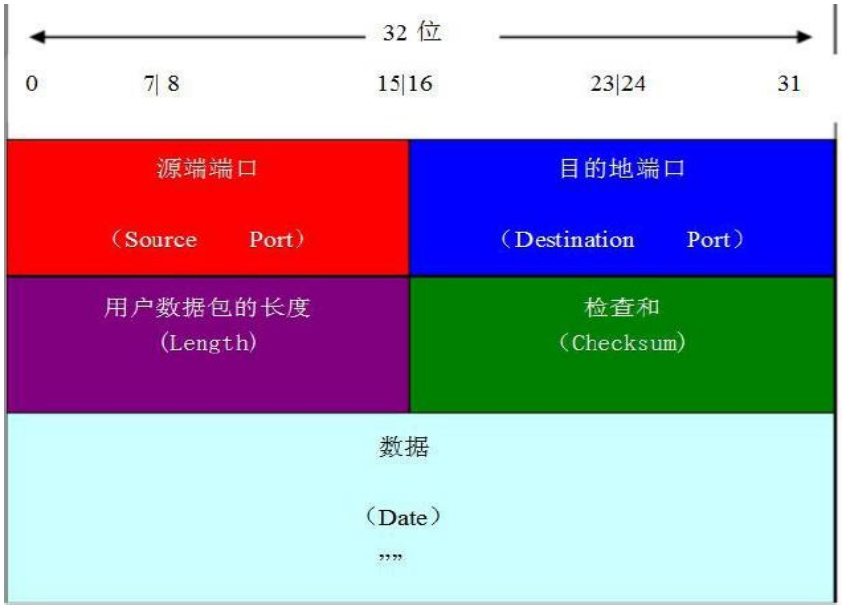
源地址和目的地址:都各占4 字节,分别记录源地址和目的地址；

12.4.4.UDP 协议

UDP 是 User Datagram Protocol（用户数据报协议）的英文缩写。UDP 只提供一种基本的、低延迟的被称为数据报的通讯。所谓数据报，就是一种自带寻址信息，从发送端走到接收端的数据包。UDP 协议经常用于图像传输、网络监控数据交换等数据传输速度要求比较高的场合。

UDP 协议的报头格式：

UDP 报头由 4 个域组成，其中每个域各占用 2 个字节，具体如下：



① UDP 源端口号

② 目标端口号

③ 数据报长度

④ 校验和

UDP 协议使用端口号为不同的应用保留其各自的数据传输通道。数据发送一方将 UDP 数据报通过源端口发送出去，而数据接收一方则通过目标端口接收数据。

数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些

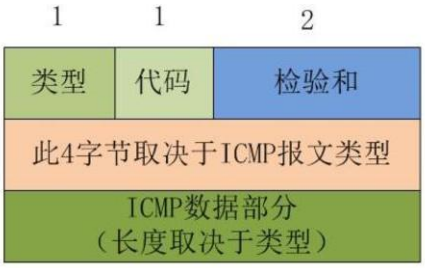
实际应用往往会限制数据报的大小，有时会降低到 8192 字节。

UDP 协议使用报头中的校验值来保证数据的安全。校验值首先在数据发送方通过特殊的算法计算得出，在传递到接收方之后，还需要再重新计算。如果某个数据报在传输过程中被第三方篡改或者由于线路噪音等原因受到损坏，发送和接收方的校验计算值将不会相符，由此 UDP 协议可以检测是否出错。虽然 UDP 提供有错误检测，但检测到错误时，错误校正，只是简单地把损坏的消息段扔掉，或者给应用程序提供警告信息。

12.4.5.Ping 功能

UDP 协议使用报头中的校验值来保证数据的安全。校验值首先在数据发送方通过特殊的算法计算得出，在传递到接收方之后，还需要再重新计算。如果某个数据报在传输过程中被第三方篡改或者由于线路噪音等原因受到损坏，发送和接收方的校验计算值将不会相符，由

此 UDP协议可以检测是否出错。虽然 UDP 提供有错误检测，但检测到错误时，错误校正，只是简单地把损坏的消息段扔掉，或者给应用程序提供警告信息。



类型值（十进制）	ICMP报文类型
0	回送应答
3	终点不可达
4	源点抑制
5	改变路由
8	回送请求
11	时间超时

12.5.SMI(MDC/MDIO)总线接口

串行管理接口（ Serial Management Interface ），也被称作 MII 管理接口（ MII ManagementInterf-ace），包括 MDC 和 MDIO 两条信号线。MDIO 是一个 PHY 的管理接口，用来读/写 PHY 的寄存器，以控制 PHY 的行为或获取 PHY 的状态，MDC 为 MDIO 提供时钟，由 MAC 端提供，在本实验中也就是 FPGA 端。在 RTL8211EG 文档里可以看到 MDC 的周期最小为 400ns，也就是最大时钟为 2.5MHz。

Table 61. MDC/MDIO Management Timing Parameters

Symbol	Description	Minimum	Maximum	Unit
t_1	MDC High Pulse Width	160	-	ns
t_2	MDC Low Pulse Width	160	-	ns
t_3	MDC Period	400	-	ns
t_4	MDIO Setup to MDC Rising Edge	10	-	ns
t_5	MDIO Hold Time from MDC Rising Edge	10	-	ns
t_6	MDIO Valid from MDC Rising Edge	0	300	ns

12.5.1.SMI 帧格式

如下图，为 SMI 的读写帧格式：

	Management Frame Fields							
	Preamble	ST	OP	PHYAD	REGAD	TA	DATA	IDLE
Read	1…1	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD	Z
Write	1…1	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD	Z

名称	说明
Preamble	由MAC 发送32 个连续的逻辑“1”，同步于MDC 信号，用于MAC 与PHY 之间的同步；
ST	帧开始位，固定为01
OP	操作码，10 表示读，01 表示写
PHYAD	PHY 的地址，5 bits
REGAD	寄存器地址，5 bits
TA	Turn Around，MDIO 方向转换，在写状态下，不需要转换方向，值为10，在读状态下，MAC 输出端为高阻态，在第二个周期，PHY 将MDIO 拉低
DATA	共16bits 数据
IDLE	空闲状态，此状态下MDIO 为高阻态，由外部上拉电阻拉高

12.5.2.读时序

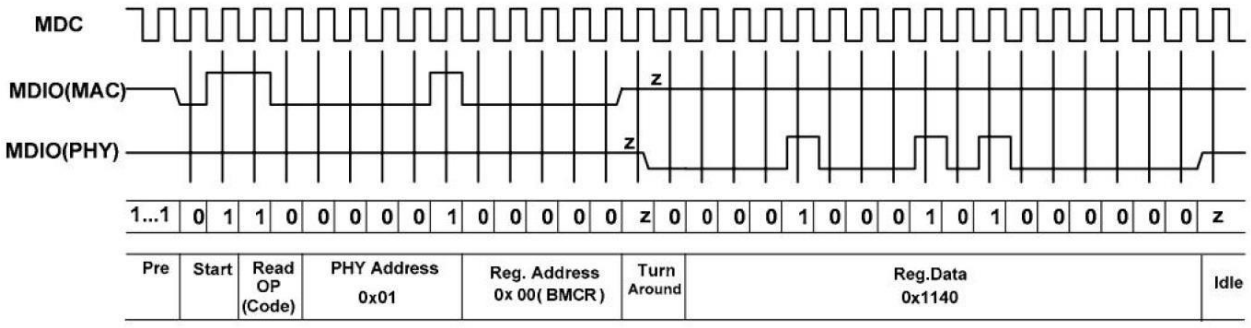


Figure 8. MDC/MDIO Read Timing

可以看到在 Turn Around 状态下，第一个周期 MDIO 为高阻态，第二个周期由 PHY 端拉

低。

12.5.3.写时序

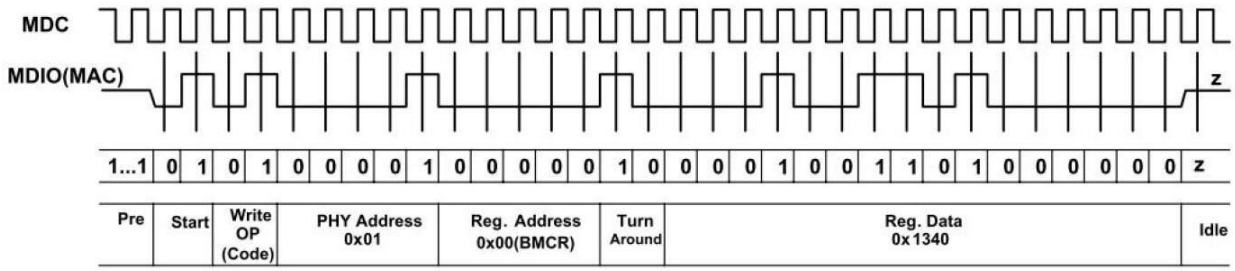


Figure 9. MDC/MDIO Write Timing

为了保证能够正确采集到数据，在 MDC 上升沿之前就把数据准备好，在本实验中为下降沿发送数据，上升沿接收数据。

12. 6. 实验设计

本实验以千兆以太网 RGMII 通信为例来设计 verilog 程序，会先发送预设的 UDP 数据到网络，每秒钟发送一次.程序分为两部分，分别为发送和接收，实现了 ARP，UDP 功能。

12.6.1.发送部分

12.6.1.1.MAC 层发送

发送部分中，mac_tx.v 为 MAC 层发送模块，首先在 SEND_START 状态，等待 mac_tx_ready信号，如果有效，表明 IP 或 ARP 的数据已经准备好，可以开始发送。再进入发送前导码状态，结束时发送 mac_data_req，请求 IP 或 ARP 的数据，之后进入发送数据状态，最后进入发送 CRC 状态。在发送数据过程中，需要同时进行 CRC 校验。前导码完成后就将上层协议数据发送出去，这个时候同样把这些上层数据放到 CRC32 模块中做序列生成，上层协议会给一个数据输出完成标志信号，这个时候 mac_tx 知道数据发送完成了，需要结束 CRC32 的序列生成，这个时候就开始提取 FCS，衔接数据之后发送出去。这样就连接了前导码---数据（Mac 帧）----FCS。之后跳转到结束状态，再回到 IDLE 状态，等待下一次的发送请求。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位

crc_result	input	32	CRC32 结果
crcen	output	1	CRC 使能信号
crcre	output	1	CRC 复位信号
crc_din	output	8	CRC 模块输入信号
mac_frame_data	input	8	从IP 或ARP 来的数据
mac_tx_req	input	1	MAC 的发送请求
mac_tx_ready	input	1	IP 或ARP 数据已准备好
mac_tx_end	input	1	IP 或ARP 数据已经传输完毕
mac_tx_data	output	8	向PHY 发送数据
mac_send_end	output	1	MAC 数据发送结束
mac_data_valid	output	1	MAC 数据有效信号，即gmii_tx_en
mac_data_req	output	1	MAC 层向IP 或ARP 请求数据

12.6.1.2.MAC 发送模式

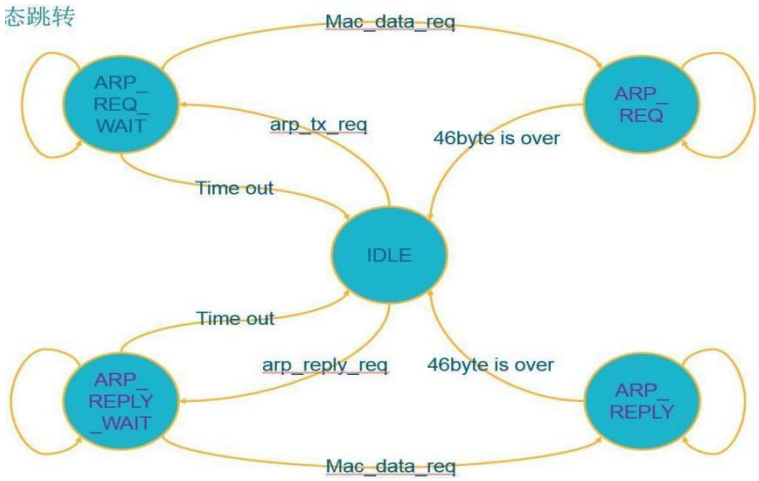
工程中的 mac_tx_mode.v 为发送模式选择，根据发送模式是 IP 或 ARP 选择相应的信号与数据。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位

mac_send_end	input	1	MAC 发送结束
arp_tx_req	input	1	ARP 发送请求
arp_tx_ready	input	1	ARP 数据已准备好
arp_tx_data	input	8	ARP 数据
arp_tx_end	input	1	ARP 数据发送到MAC 层结束
arp_tx_ack	input	1	ARP 发送响应信号
ip_tx_req	input	1	IP 发送请求
ip_tx_ready	input	1	IP 数据已准备好
ip_tx_data	input	8	IP 数据
ip_tx_end	input	1	IP 数据发送到MAC 层结束
mac_tx_ready	output	1	MAC 数据已准备好信号
ip_tx_ack	output	1	IP 发送响应信号
mac_tx_ack	input	1	MAC 发送响应信号
mac_tx_req	output	1	MAC 发送请求
mac_tx_data	output	8	MAC 发送数据
mac_tx_end	output	1	MAC 数据发送结束

12.6.1.3.ARP 发送

发送部分中，arp_tx.v 为 ARP 发送模块， 在 IDLE 状态下，等待 ARP 发送请求或ARP应答请求信号，之后进入请求或应答等待状态，并通知 MAC 层，数据已经准备好，等待 mac_data_req 信号，之后进入请求或应答数据发送状态。由于数据不足 46 字节，需要补全46 字节发送。



信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
destination_mac_addr	input	48	发送的目的MAC 地址
source_mac_addr	input	48	发送的源MAC 地址
source_ip_addr	input	32	发送的源IP 地址
destination_ip_addr	input	32	发送的目的IP 地址
mac_data_req	input	1	MAC 层请求数据信号
arp_request_req	input	1	ARP 请求的请求信号

arp_reply_ack	output	1	ARP 回复的应答信号
arp_reply_req	input	1	ARP 回复的请求信号
arp_rec_source_ip_addr	input	32	ARP 接收的源IP 地址，回复时放到目的IP地址
arp_rec_source_mac_addr	input	48	ARP 接收的源MAC 地址，回复时放到目的MAC 地址
mac_send_end	input	1	MAC 发送结束
mac_tx_ack	input	1	MAC 发送应答
arp_tx_ready	output	1	ARP 数据准备好
arp_tx_data	output	8	ARP 发送数据
arp_tx_end	output	1	ARP 数据发送结束
arp_tx_req	output	1	ARP 发送请求信号

12.6.1.4.IP 层发送

在发送部分，ip_tx.v 为 IP 层发送模块，在 IDLE 状态下，如果 ip_tx_req 有效，也就是 UDP 或 ICMP 发送请求信号，进入等待发送数据长度状态，之后进入产生校验和状态，校验和是将 IP 首部所有数据以 16 位相加，最后将进位再与低 16 位相加，直到进入为 0，再将低

16 位取反，得出校验和结果。

在生成校验和之后，等待 MAC 层数据请求，开始发送数据，并在即将结束发送 IP 首部后请求 UDP 或 ICMP 数据。等发送完，进入 IDLE 状态。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rstn	input	1	低电平复位
dest_mac_addr	input	48	发送的目的 MAC 地址
sour_mac_addr	input	48	发送的源 MAC 地址
sour_ip_addr	input	32	发送的源 IP 地址
dest_ip_addr	input	32	发送的目的 IP 地址
ttl	input	8	生存时间
ip_send_type	input	8	上层协议号，如 UDP,ICMP
upper_layer_data	output	8	从UDP 或ICMP 过来的数据
upper_data_req	input	1	向上层请求数据
mac_tx_ack	input	1	MAC 发送应答
mac_send_end	input	1	MAC 发送结束信号
mac_data_req	input	1	MAC 层请求数据信号
upper_tx_ready	input	1	上层UDP 或ICMP 数据准备好
ip_tx_req	input	1	发送请求，从上层过来
ip_send_data_length	input	16	发送数据总长度

ip_tx_ack	output	1	产生 IP 发送应答
ip_tx_ready	output	1	IP 数据已准备好
ip_tx_data	output	8	IP 数据
ip_tx_end	output	1	IP 数据发送到 MAC 层结束

12.6.1.5.IP 发送模式

工程中的 ip_tx_mode.v 为发送模式选择，根据发送模式是 UDP 或 ICMP 选择相应的信号与数据。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
mac_send_end	input		MAC 数据发送结束
udp_tx_req	input	1	UDP 发送请求
udp_tx_ready	input	1	UDP 数据准备好
udp_tx_data	input	8	UDP 发送数据
udp_send_data_length	input	16	UDP 发送数据长度
udp_tx_ack	output	1	输出UDP 发送应答
icmp_tx_req	input	1	ICMP 发送请求
icmp_tx_ready	input	1	ICMP 数据准备好

icmp_tx_data	input	8	ICMP 发送数据
icmp_send_data_length	input	16	ICMP 发送数据长度
icmp_tx_ack	output	1	ICMP 发送应答
ip_tx_ack	input	1	IP 发送应答
ip_tx_req	input	1	IP 发送请求
ip_tx_ready	output	1	IP 数据已准备好
ip_tx_data	output	8	IP 数据
ip_send_type	output	8	上层协议号，如UDP,ICMP
ip_send_data_length	output	16	发送数据总长度

12.6.1.6.UDP 发送

发送部分中，udp_tx.v 为 UDP 发送模块。

信号名称	方向	位宽	说明
udp_send_clk	input	1	系统时钟
rstn	input	1	低电平复位
app_data_in_valid	input	1	从外部所接收的数据输出有效信号
app_data_in	input	8	外部所接收的数据
app_data_length	input	16	从外部所接收的当前数据包的长度（不含 udp、ip、mac 首部）

udp_dest_port	input	16	从外部所接收的数据包的源端口号
app_data_request	input	1	用户接口数据发送请求
udp_send_ready	output	1	UDP 数据发送准备
udp_send_ack	output	1	UDP 数据发送应当
ip_send_ready	input	1	IP 数据发送准备
ip_send_ack	input	1	IP 数据发送应当
udp_send_request	output	1	用户接口数据发送请求
udp_data_out_valid	output	1	发送的数据输出有效信号
udp_data_out	output	8	发送的数据输出
udp_packet_length	output	16	当前数据包的长度（不含 udp、 ip、mac 首部）

12.6.2.接收部分

12.6.2.1.MAC 层接收

在接收部分，其中 mac_rx.v 为 mac 层接收文件，首先在 IDLE 状态下当 rx_en 信号为高，进入 REC_PREAMBLE 前导码状态，接收前导码。之后进入接收 MAC 头部状态，即目的 MAC地址，源 MAC 地址，类型，将它们缓存起来，并在此状态判断前导码是否正确，错误则进入REC_ERROR 错误状态，在 REC_IDENTIFY 状态判断类型是 IP（8'h0800）或 ARP(8'h0806)。然后进入接收数据状态，将数据传送到 IP 或 ARP 模块，等待 IP 或 ARP 数据接收完毕，再接收 CRC 数据。并在接收数据的过程中对接收的数据进行 CRC 处理，将结果与接收到的 CRC 数据进行对比，判断数据是否接收正确，正确则结束，错误则进入 ERROR 状态。

信号名称	方向	位宽	说明

clk	input	1	系统时钟
rstn	input	1	低电平复位
rx_en	input	1	开始接受使能
mac_rx_datain	input	8	接受的数据
checksum_err	input	1	IP 层校验错误信号
ip_rx_end	input	1	IP 接受结束
arp_rx_end	input	1	ARP 接受结束
ip_rx_req	output	1	IP 接受请求
arp_rx_req	input	1	请求 ARP 接收
mac_rx_dataout	output	8	MAC 层接收数据输出给 IP 或 ARP
mac_rec_error	output	1	MAC 层接收错误
mac_rx_dest_mac_addr	output	48	MAC 接收的目的 IP 地址
mac_rx_sour_mac_addr	output	48	MAC 接收的源 IP 地址

12.6.2.2.ARP 接收

工程中的 arp_rx.v 为 ARP 接收模块，实现 ARP 数据接收，在 IDLE 状态下，接收到从 MAC层发来的 arp_rx_req 信号，进入 ARP 接收状态，在此状态下，提取出目的 MAC 地址，源 MAC地址，目的 IP 地址，源 IP 地址，并判断操作码 OP 是请求还是应答。如果是请求，则判断接收到的目的 IP 地址是否为本机地址，如果是，发送应答请求信号 arp_reply_req，如果不是，则忽略。如果 OP 是应答，则判断接收到的目的 IP 地址及目的

MAC地址是否与本机一致，如果是，则拉高arp_found信号，表明接收到了对方的地址。并将对方的MAC 地址及IP地址存入ARP缓存中。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rstn	input	1	低电平复位
local_ip_addr	input	32	本地 IP 地址
local_mac_addr	input	48	本地 MAC 地址
arp_rx_data	input	8	ARP 接收数据
arp_rx_req	input	1	ARP 接收请求
arp_rx_end	output	1	ARP 接收完成
arp_reply_ack	input	1	ARP 回复应答
arp_reply_req	output	1	ARP 回复请求
arp_rec_sour_ip_addr	input	32	ARP 接收的源 IP 地址
arp_rec_sour_mac_addr	input	48	ARP 接收的源 MAC 地址
arp_found	output	1	ARP 接收到请求应答正确

12.6.2.3.IP 层接收模块

在工程中，ip_rx 为IP层接收模块，实现IP层的数据接收，信息提取，并进行校验和检查。首先在IDLE状态下，判断从MAC层发过来的ip_rx_req信号，进入接收IP首部状态，先在REC_HEADER0提取出首部长度及IP总长度，进入REC_HEADER1状态，在此状态提取出目

的IP地址，源IP地址，协议类型，根据协议类型发送udp_rx_req或icmp_rx_req。在接收首部的同时进行校验和的检查，将首部接收的所有数据相加，存入32位寄存器，再将高16位

与低16位相加，直到高16位为0 ，再将低16位取反，判断其是否为0，如果是0，则检验正确，否则错误，进入IDLE状态，丢弃此帧数据，等待下次接收。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
local_ip_addr	input	32	本地IP 地址
local_mac_addr	input	48	本地MAC 地址
ip_rx_data	input	8	从MAC 层接收的数据
ip_rx_req	input	1	MAC 层发送的IP 接收请求信号
mac_rx_destination_mac_addr	input	48	MAC 层接收的目的MAC 地址
udp_rx_req	output	1	UDP 接收请求信号
icmp_rx_req	output	1	ICMP 接收请求信号
ip_addr_check_error	output	1	地址检查错误信号
upper_layer_data_length	output	16	上层协议的数据长度
ip_total_data_length	output	16	数据总长度
net_protocol	output	8	网络协议号
ip_rec_source_addr	output	32	IP 层接收的源IP 地址

ip_rec_destination_addr	output	32	IP 层接收的目的IP 地址
ip_rx_end	output	1	IP 层接收结束
ip_checksum_error	output	1	IP 层校验和检查错误信号

12.6.2.4.UDP 接收

在工程中，udp_rx.v为UDP 接收模块，在此模块首先接收UDP首部，再接收数据部分，在接收的同时进行UDP校验和检查，如果UDP数据是奇数个字节，在计算校验和时，在最后一个字节后加上8'h00，并进行校验和计算。校验方法与IP校验和一样，如果校验正确，将拉高udp_rec_data_valid 信号，表明接收的UDP数据有效，否则无效，等待下次接收。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
udp_rx_data	input	8	UDP 接收数据
udp_rx_req	input	1	UDP 接收请求
mac_rec_error	input	1	MAC 层接收错误
net_protocol	input	8	网络协议号
ip_rec_source_addr	input	32	IP 层接收的源IP 地址
ip_rec_destination_addr	input	32	IP 层接收的目的IP 地址
ip_checksum_error	input	1	IP 层校验和检查错误信号

ip_addr_check_error	input	1	地址检查错误信号
upper_layer_data_length	input	16	上层协议的数据长度
udp_rec_ram_rdata	output	8	UDP 接收RAM 读数据
udp_rec_ram_read_addr	input	11	UDP 接收RAM 读地址
udp_rec_data_length	output	16	UDP 接收数据长度
udp_rec_data_valid	output	1	UDP 接收数据有效

12.6.3.其他部分

12.6.3.1.ICMP 应答

在工程中，icmp_reply.v 实现ping功能，首先接收其他设备发过来的icmp数据，判断类型是否是回送请求（ECHO REQUEST），如果是，将数据存入RAM，并计算校验和，判断校验和是否正确，如果正确则进入发送状态，将数据发送出去。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
mac_send_end	input	1	Mac 发送结束信号
ip_tx_ack	input	1	IP 发送应答
icmp_rx_data	input	8	ICMP 接收数据
icmp_rx_req	input	1	ICMP 接收请求
icmp_rev_error	input	1	接收错误信号

upper_layer_data_length	input	16	上层协议长度
icmp_data_req	output	1	发送请求ICMP 数据
icmp_tx_ready	output	1	ICMP 发送准备好
icmp_tx_data	output	8	ICMP 发送数据
icmp_tx_end	output	1	ICMP 发送结束
icmp_tx_req	output	1	ICMP 发送请求

12.6.3.2.ARP 缓存

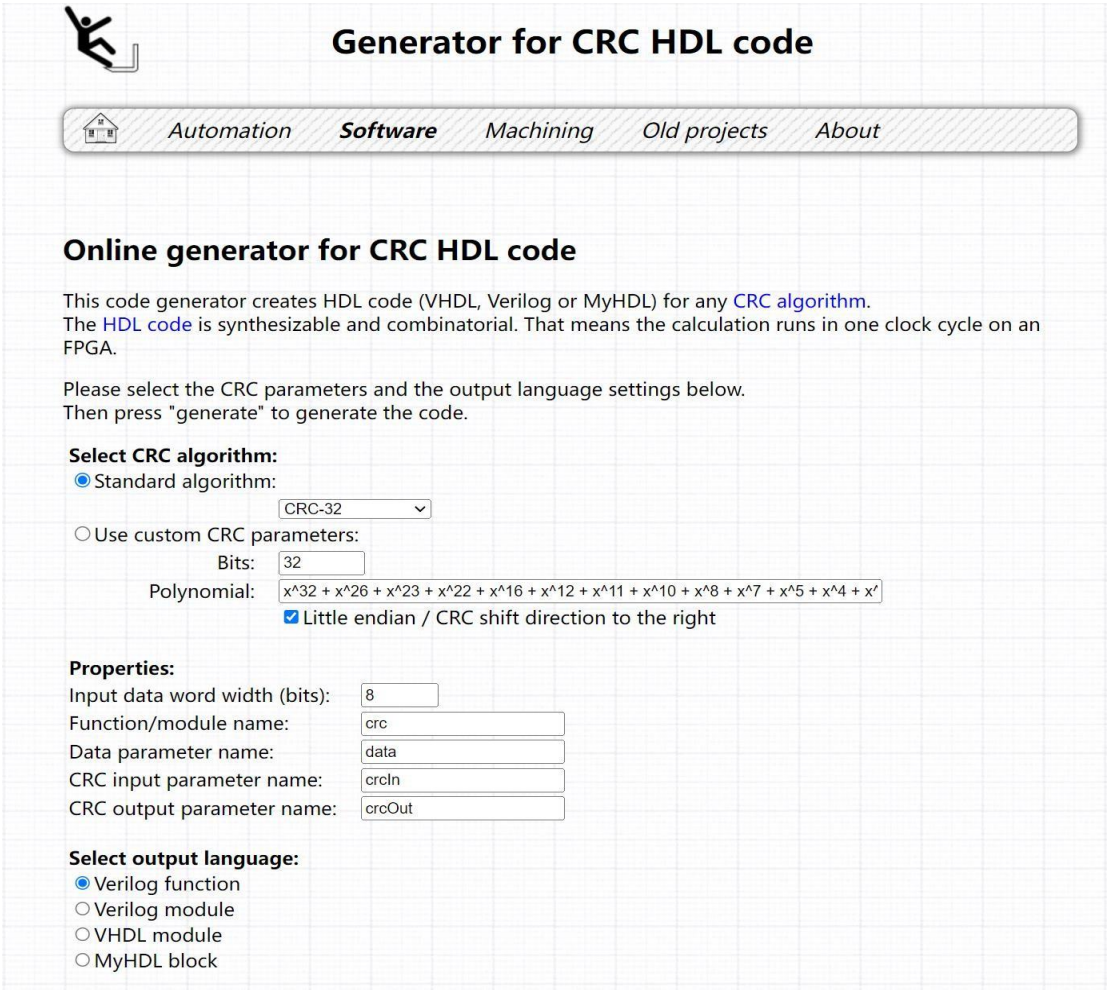
在工程中，arp_cache.v 为 arp 缓存模块，将接收到的其他设备 IP 地址和MAC 地址缓存，在发送数据之前，查询目的地址是否存在，如果不存在，则向目的地址发送 ARP 请求，等待应答。在设计文件中，只做了一个缓存空间，如果有需要，可扩展。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
arp_found	input	1	ARP 接收到回复正确
arp_rec_source_ip_addr	input	32	ARP 接收的源IP 地址
arp_rec_source_mac_addr	input	48	ARP 接收的源MAC 地址
destination_ip_addr	input	32	目的IP 地址
destination_mac_addr	output	48	目的MAC 地址

mac_not_exist	output	1	目的地址对应的MAC 地址不存在

12.6.3.3.CRC 校验模块(crc.v)

CRC32 校验是在目标MAC地址开始计算的，一直计算到一个包的最后一个数据为止。一些网站可以自动生成CRC算法的verilog文件：https://bues.ch/cms/hacking/crcgen.html



12.6.4.实验现象

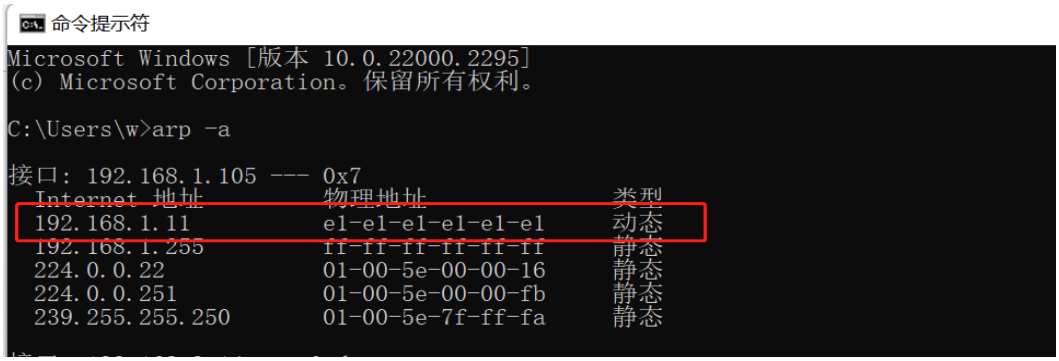
用网线连接 MES50HP 开发板网口 1 和 PC 端网口；设置接收端（PC 端）IP 地址为 192.168.1.105，开发板的 IP 地址为 192.168.1.11 如下图：

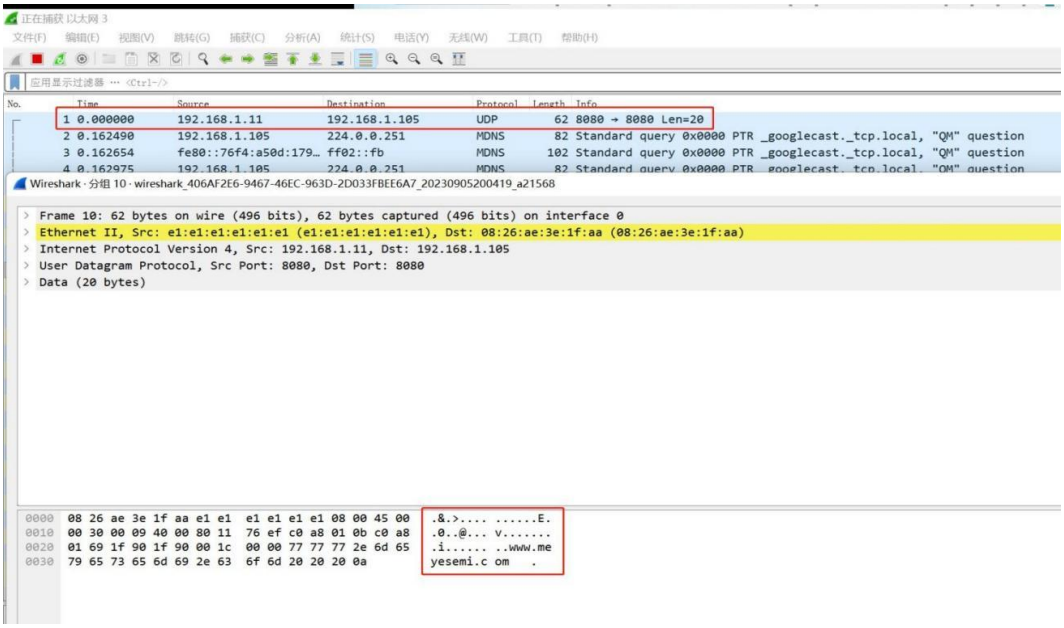
```
module ethernet1_test#(
    parameter       LOCAL_MAC = 48'he1_e1_e1_e1_e1_e1,
    parameter       LOCAL_IP  = 32'hC0_A8_01_0B,//192.168.1.11
    parameter       LOCL_PORT = 16'h1F90,

    parameter       DEST_IP   = 32'hC0_A8_01_69,//192.168.1.105
    parameter       DEST_PORT = 16'h1F90
)(
```

将程序下载到开发板后，便可以看到 LED 灯规律性闪烁：

信号名称	LED编号	参考说明
led	1	闪烁

通过命令提示符，输入 arp -a，可以查到 IP：192.168.1.11MAC：e1_e1_e1_e1_e1_e1;





成功建立连接后会持续发送数据报“www.meyesemi.com”

13.光纤通信测试实验例程

13. 1. 实验简介

实验目的:

通过光纤连接实现光模块之间的数据收发。

实验环境：

Window11

PDS2022.2

硬件环境：

OPHW-25开发板

13. 2. 实验原理

OPHW-25内置了线速率高达6.6Gbps 高速串行接口模块,即HSSTLP,包含1个HSSTLP，共4个全双工收发LANE，除了PMA，HSSTLP 还集成了丰富的PCS 功能，可灵活应用于各种串行协议标准。在产品内部,每个HSST 支持1~4 个全双工收发LANE。HSST 主要特性包括：

- 支持 DataRate 速率：0.6Gbps-6.6Gbps
- 灵活的参考时钟选择方式
- 发送通道和接收通道数据率可独立配置
- 可编程输出摆幅和去加重
- 接收端自适应线性均衡器 Logos2系列FPGA器件数据手册
- PMA Rx 支持 SSC
- 数据通道支持数据位宽：8bit only, 10bit only, 8b10b, 16bit only, 20bit only, 32bit only, 40bit only,64b66b/64b67b 等模式
- 可灵活配置的 PCS，可支持 PCI Express GEN1, PCI Express GEN2,XAUI,千兆以太网,CPRI,SRIO 等协议
- 灵活的 Word Alignment 功能

- 支持 RxClock Slip 功能以保证固定的 Receive Latency
- 支持协议标准 8b10b 编码解码
- 支持协议标准 64b66b/64b67b 数据适配功能
- 灵活的 CTC 方案
- 支持 x2 和 x4 的 Channel Bonding
- HSSTLP 的配置支持动态修改
- 近端环回和远端环回模式
- 内置 PRBS 功能
- 自适应

13. 3. 工程说明

13.3.1.安装 HSST IP 核

PDS安装后，需手动添加HSST IP，请按以下步骤完成：

（1）HSST IP文件：选择1_9.iar

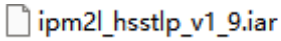


图 13.3-1

（2）IP安装步骤：请查看 “工具使用篇\03_IP核安装与查看用户指南”

图 13.3-2

13.3.2. 光纤通信测试例程

打开PDS软件，新建工程hsst_test，点开如下图标，打开IP Compiler；

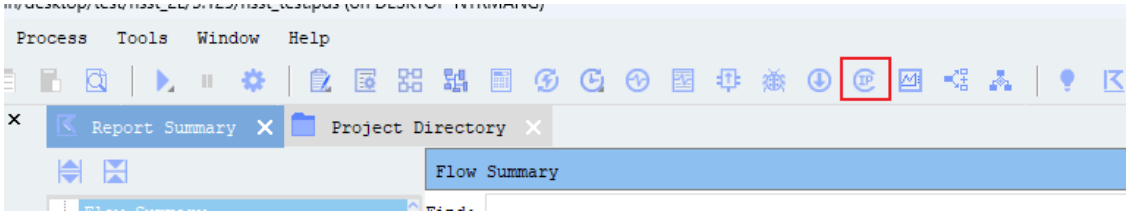


图 13.3-3

选择HSST IP，取名，然后点击Customize；

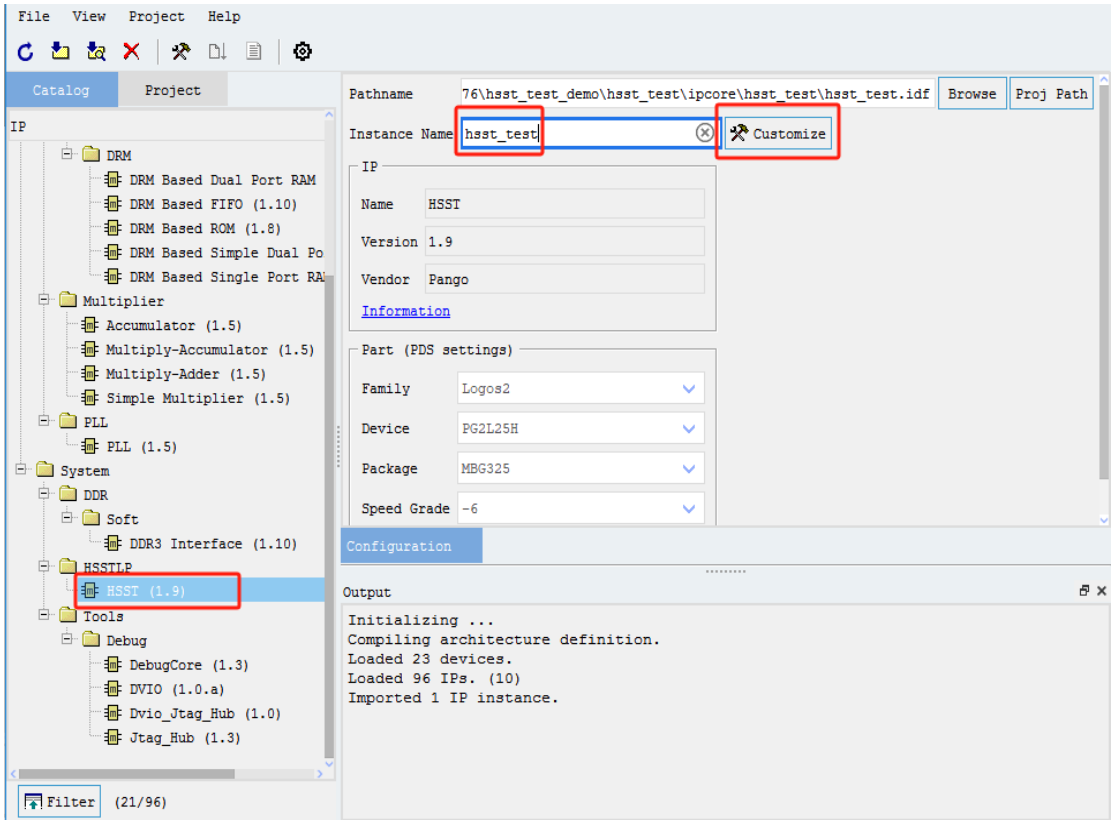


图 13.3-4

在HSST设置界面中Protocol and Rate按照如下设置，Channel0 Channel1为DISABLE，Channel2 Channel3为Fullduplex：

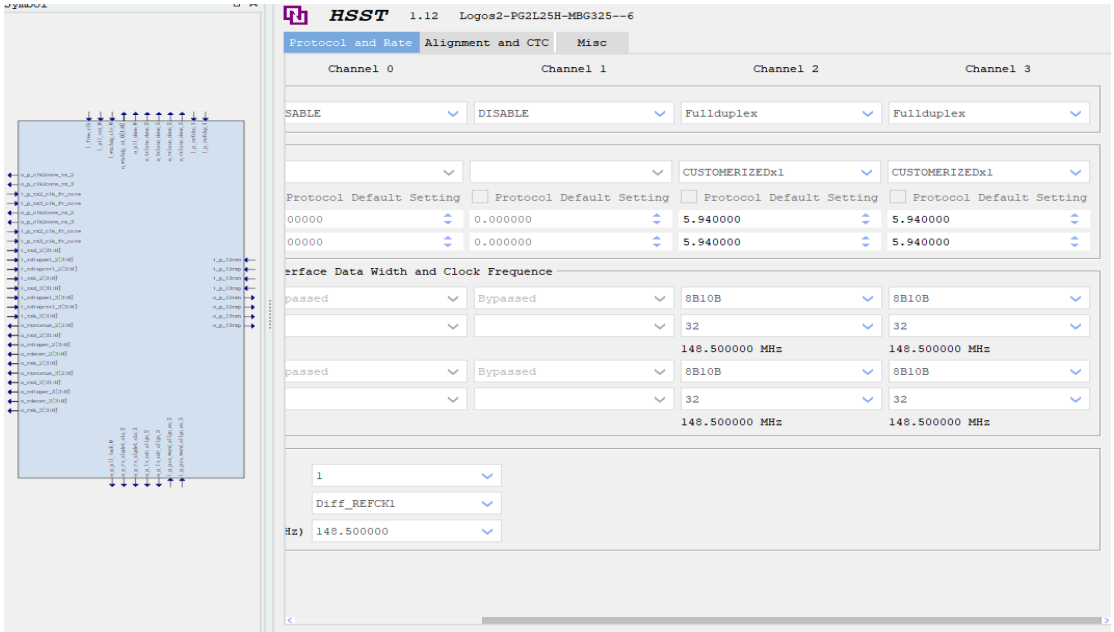


图 13.3-5

Alignment and CTC按照如下设置：

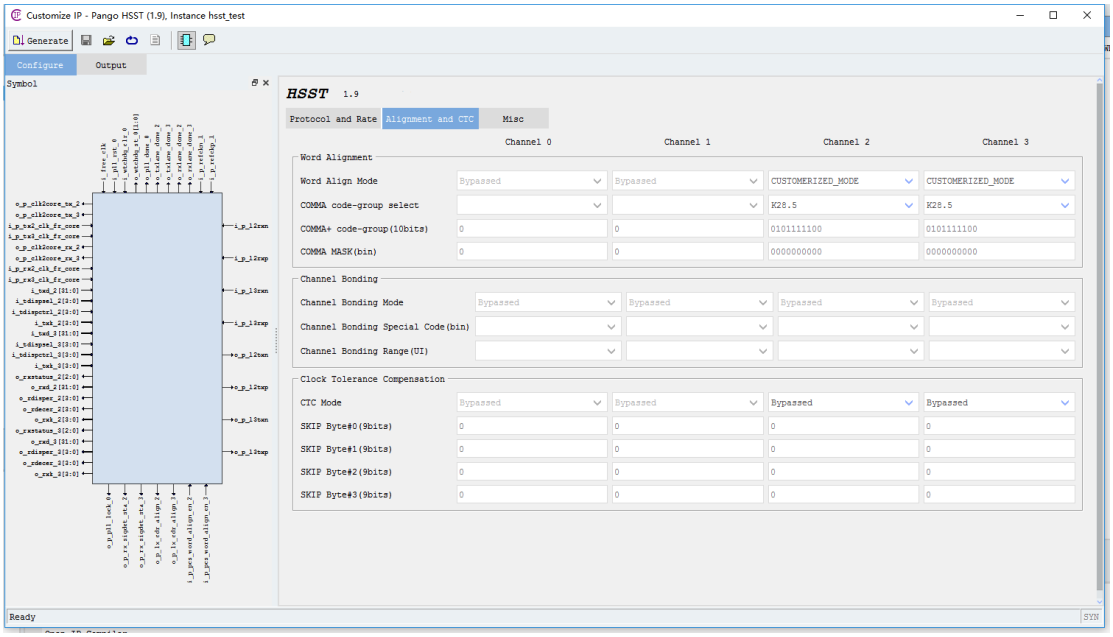


图 13.3-6

Misc按照如下设置，点击Generate可生成HSST IP；

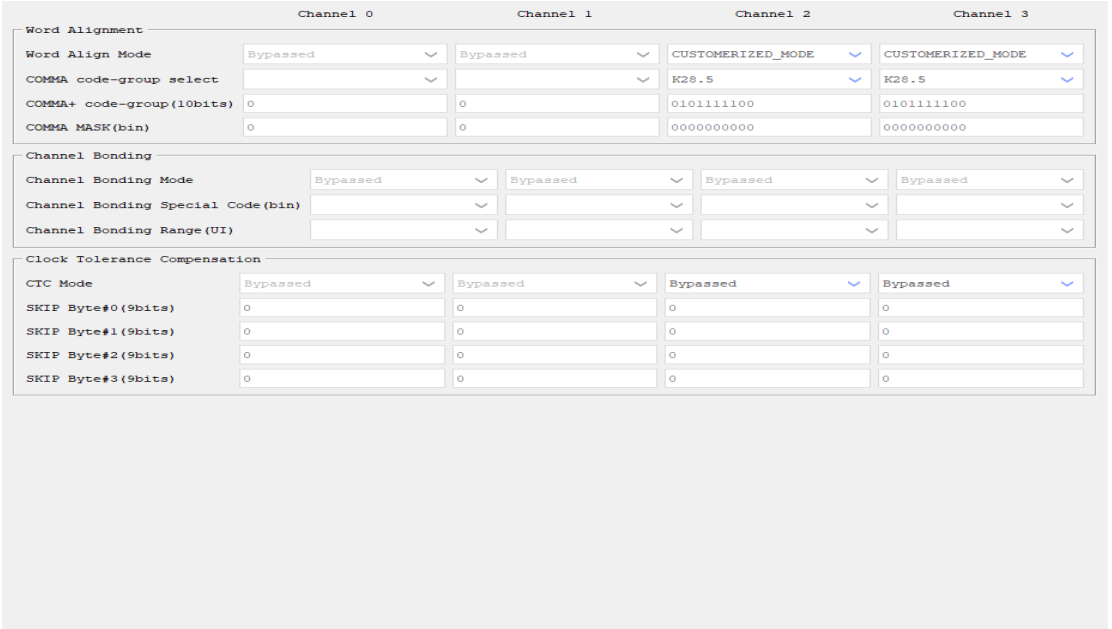


图 13.3-7

关闭本工程，在IP保存路径下打开IP Example工程：

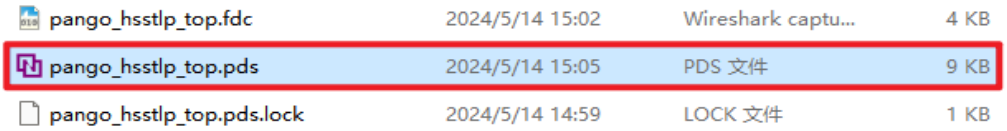


图 13.3-8

为了能在开发板上运行，需对顶层文件hsst_test_dut_top的复位进行修改，详情请查看例程顶层文件：

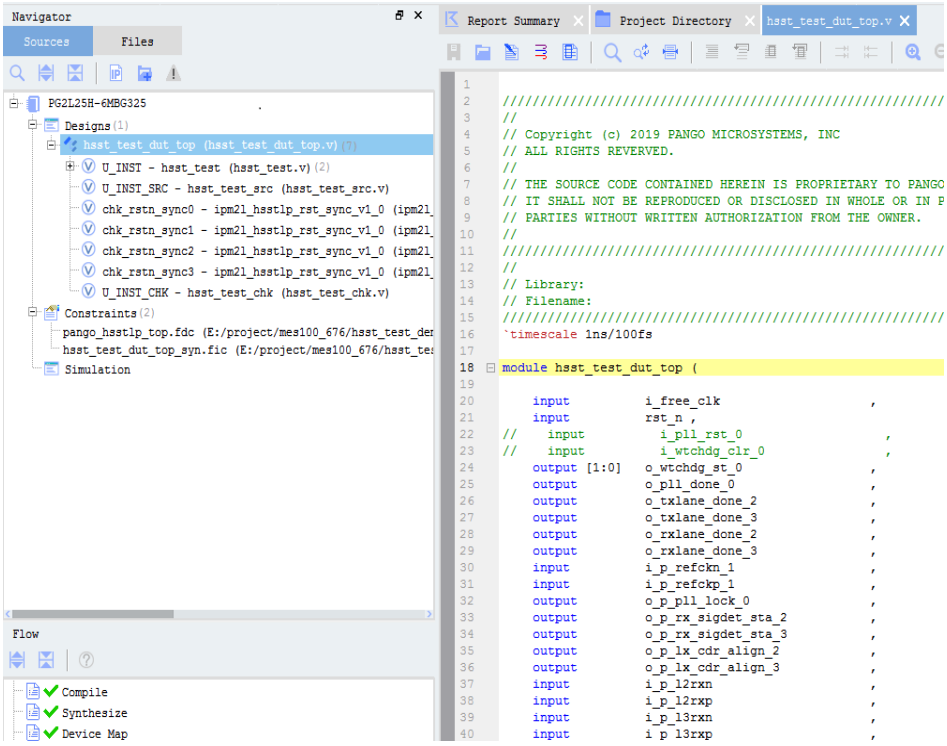


图 13.3-9

修改管脚分配，详情请查看原理图或10_hsst_test例程；

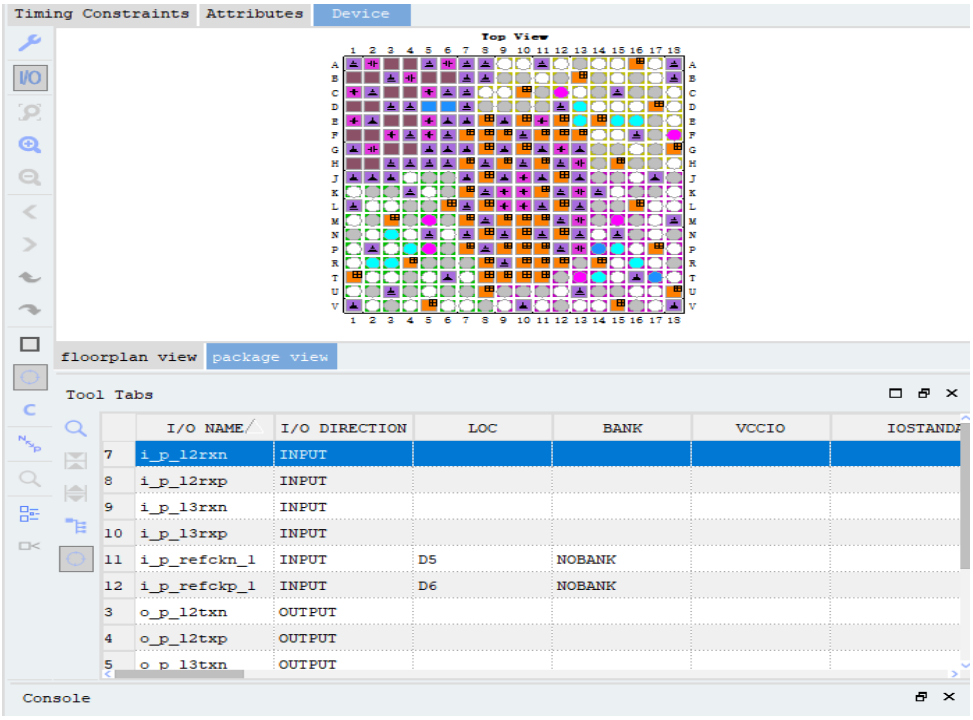


图 13.3-10

进行Debugger插核操作。

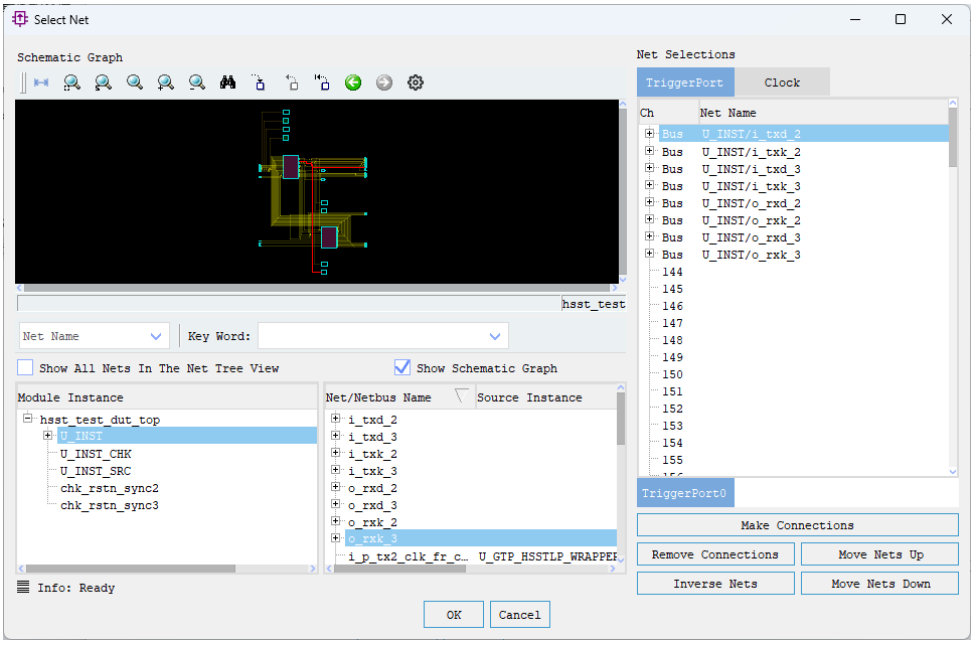


图 13.3-11

可按以下方式查看IP核的用户指南，了解Example模块组成；

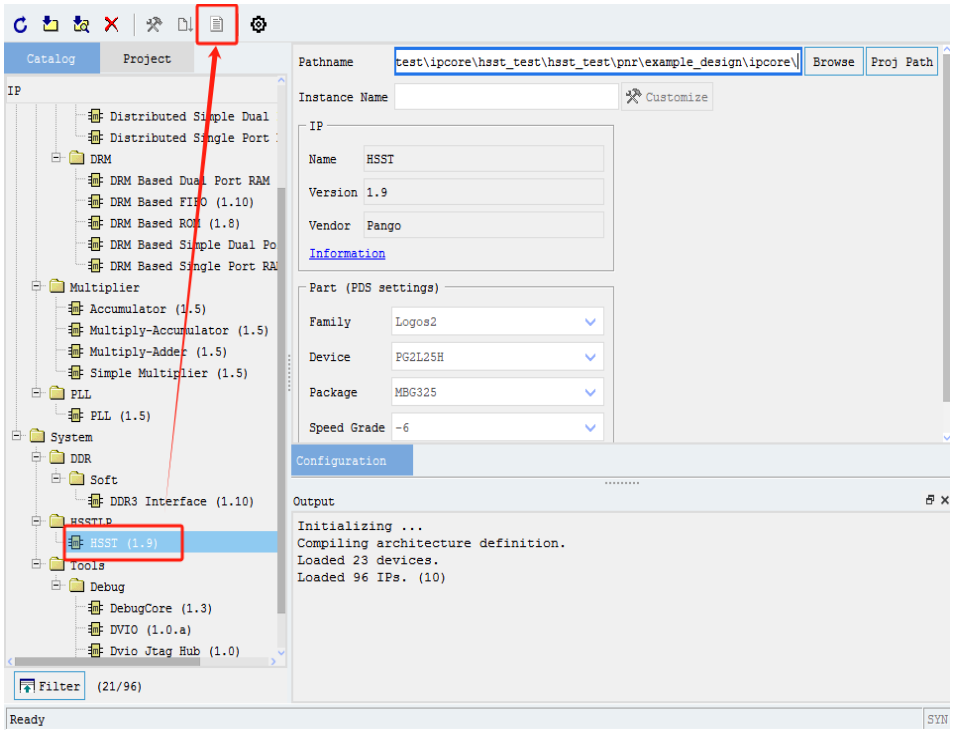


图 13.3-12

13.4. 实验现象

注：例程位置：2_Demo\hsst_test\ipcore\hsst_test\pnr\example_design

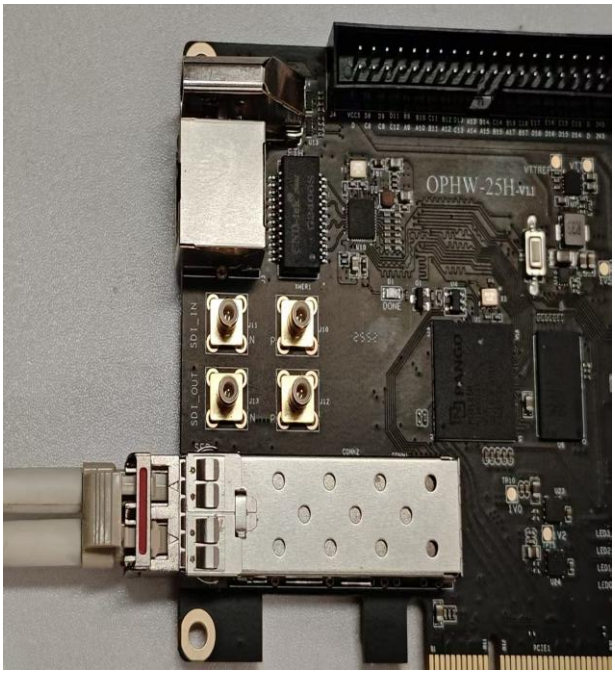


图 13.4-1

把光纤两端接入SFP0\接口（用户需购买光模块），进行Debugger在线调试，可看到窗口中发送和接收的数据一致的。

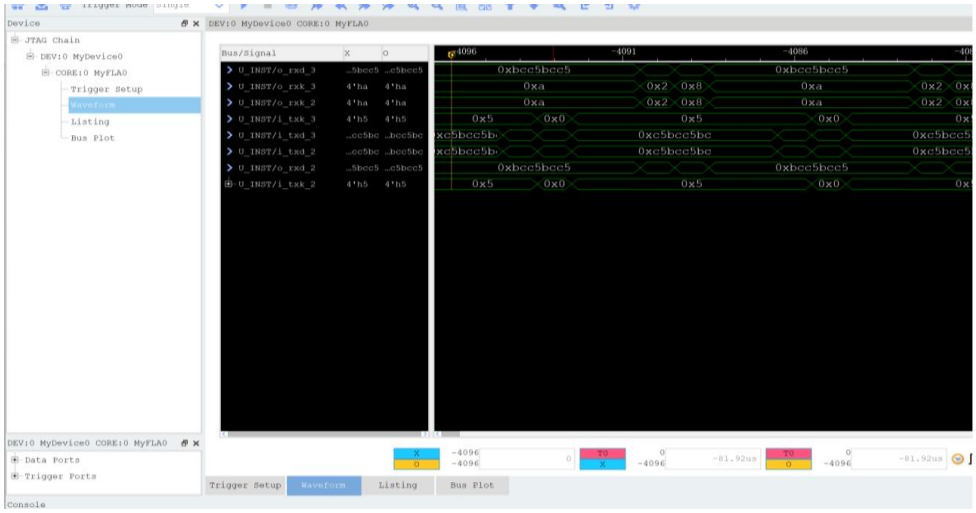


图 13.4-2

说明：

K码对应的是bc，当该字节为bc时，K码为1。所以当K码变化规律固定，且数据只出现移位时，数据是正确的。

例如收到的数据是0xbcc5bcc5，该数据为32bit，对应4字节，与rxk相对应，所以此时bc码出现在第2个字节和第4个字节。所以对应的o_rxk为4'b1010即16进制为4'ha，也就是Debugger显示的0xa。

2、例程文件的管脚约束需要将这里保留

```
create_clock -name {i_p_refckn_1} [get_ports i_p_refckn_1] -period {6.700} -waveform {0.000 3.350}
create_clock -name {i_p_refckp_1} [get_ports i_p_refckp_1] -period {6.700} -waveform {0.000 3.350}


define_attribute {i:u_hsst_test.U_GTP_HSSTLP_WRAPPER.CHANNEL2_ENABLE.U_GTP_HSSTLP_LANE2} {PAP_LOC} {HSSTLP_268_306:U2_HSSTLP_LANE}
define_attribute {i:u_hsst_test.U_GTP_HSSTLP_WRAPPER.CHANNEL3_ENABLE.U_GTP_HSSTLP_LANE3} {PAP_LOC} {HSSTLP_268_306:U3_HSSTLP_LANE}
define_attribute {i:u_hsst_test.U_GTP_HSSTLP_WRAPPER.PLL0_ENABLE.U_GTP_HSSTLP_PLL0} {PAP_LOC} {HSSTLP_268_306:U1_HSSTLP_PLL}

define_attribute {p:i_p_refckn_1} {PAP_IO_LOC} {D5}
define_attribute {p:i_p_refckp_1} {PAP_IO_LOC} {D6}
define_attribute {p:sys_clk} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:sys_clk} {PAP_IO_LOC} {P14}
define_attribute {p:sys_clk} {PAP_IO_VCCIO} {3.3}
define_attribute {p:sys_clk} {PAP_IO_STANDARD} {LVCMOS33}
define_attribute {p:sys_clk} {PAP_IO_UNUSED} {TRUE}
define_attribute {p:tx_disable[0]} {PAP_IO_DIRECTION} {OUTPUT}
define_attribute {p:tx_disable[0]} {PAP_IO_LOC} {T17}
define_attribute {p:tx_disable[0]} {PAP_IO_VCCIO} {3.3}
define_attribute {p:tx_disable[0]} {PAP_IO_STANDARD} {LVCMOS33}
define_attribute {p:tx_disable[0]} {PAP_IO_DRIVE} {4}
define_attribute {p:tx_disable[0]} {PAP_IO_NONE} {TRUE}
define_attribute {p:tx_disable[0]} {PAP_IO_SLEW} {FAST}
```

图 13.4-3

14.PCIE 通信测试实验例程

14.1. 实验简介

实验目的：

完成PCIE通信测试。

实验环境：

Window11

PDS2022.2

硬件环境：

OPHW-25开发板

14.2. 实验原理

OPHW-25集成内置了线速率高达6.6Gbps 高速串行接口模块，即HSSTLP。OPHW-25开发板提供一个PCIe x2 接口，PCIE卡的外形尺寸符合标准PCIe 卡电气规范要求，可直接在普通PC 的x2 PCIe插槽上使用。

14.3. PCIE 简介

PCIE IP符合PCI Express® Base Specification Revision 2.1[8]协议和PHY Interface for the PCI ExpressTM Architecture Version 2.00[12]（数据通路扩展为32 bits）协议。

功能特性	特性说明
支持配置Device Type	PCI Express Endpoint
	Legacy PCI Express Endpoint
	Root Port of PCI Express Root Complex
支持配置Max Link Width	x1
	x2
	x4
支持配置Max Link Speed	2.5GT/s
	5GT/s
支持选择Reference Clk频率	100MHz
	125MHz
支持Upconfigure Capable	-
支持Debug接口	-

支持通过Apb动态配置PCIe Configuration Space	
支持Receive Queue Management	-
支持Lane Reversal	-
支持Force No Scrambling	-
支持配置ID	支持配置Vendor ID
	支持配置Device ID
	支持配置Revision ID
	PCI Express Endpoint、Legacy PCI Express Endpoint支持配置Subsystem Vendor ID
	PCI Express Endpoint、Legacy PCI Express Endpoint支持配置Subsystem ID
	支持配置Classcode
支持BAR配置	PCI Express Endpoint、Legacy PCI Express Endpoint支持配置6个BAR
	Root Port of PCI Express Root Complex仅支持配置BAR0、BAR1
	PCI Express Endpoint支持配置为Memory BAR
	Legacy PCI Express Endpoint、Root Port of PCI Express Root Complex支持配置为Memory、IO BAR
	支持32bit BAR
	32bit BAR支持配置大小为256 Byte -2G Byte
	BAR0、BAR2、BAR4支持64bit BAR
	64bit BAR支持Prefetchable
	64bit BAR支持配置大小为256 Byte -8E Byte
	支持Expansion ROM BAR
	Expansion ROM BAR支持配置大小为2K Byte -16M Byte
支持配置Max Payload Size	128 Byte
	256 Byte
	512 Byte

	1024 Byte
支持配置Extended Tag Field与Extended Tag Default	-
支持Atomic事务	-
RC 时 支 持 设 置 Read Completion Boundary	-
支持配置Target Link Speed	-
RC时支持设置CRS Software Visibility	-
支 持 设 置 ECRC Generation Capable	-
功能特性	特性说明
默认使能ECRC Check Capable	-
支持INIT中断	PCI Express Endpoint、Legacy PCI Express Endpoint只支持INTA
	Root Port of PCI Express Root Complex支持INTA、INTB、INTC、INTD
支持MSI中断	支持64-bit Address MSI中断
	支持Multiple Message Capable：1、2、4、8、16、32个Vectors
	支持Per Vector Masking Capable
支持MSIx中断	支持配置Table Size 、Offset与BIR
	支持配置PBA Offset与BIR

14. 4. 工程说明

14.4.1. 安装 PCIE IP 核

PDS安装后，需手动添加PCIE IP，请按以下步骤完成：

PCIE IP文件：6_IP_setup_packet\ips2l_pcie_gen2_v1_2c.iar

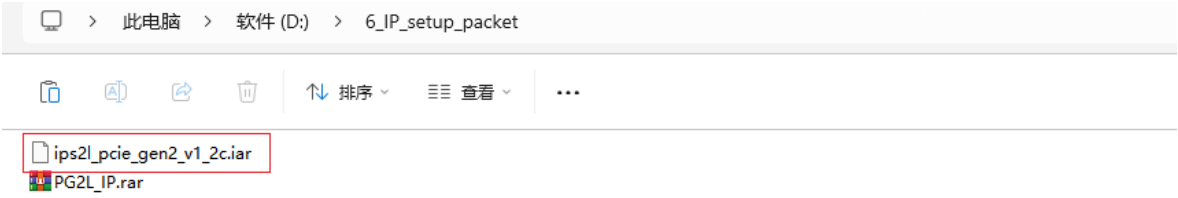


图 14.4-1

IP安装步骤：请查看 工具使用篇\03_IP核安装与查看用户指南

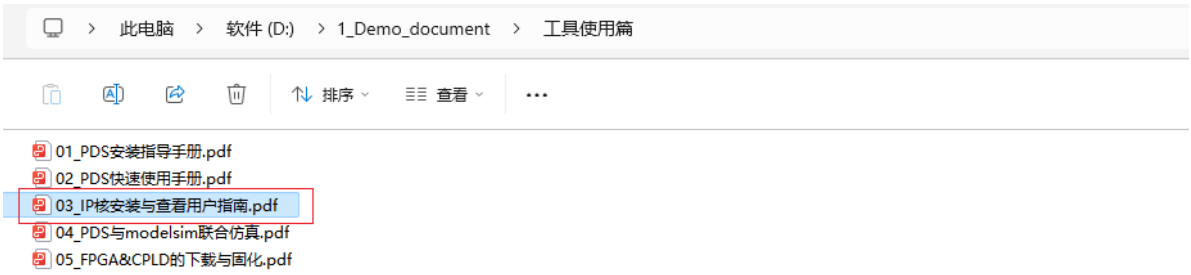


图 14.4-2

14.4.2. PCIE 参考设计例程

打开PDS软件，新建工程pcie_test，点开如下图标，打开IP Compiler；

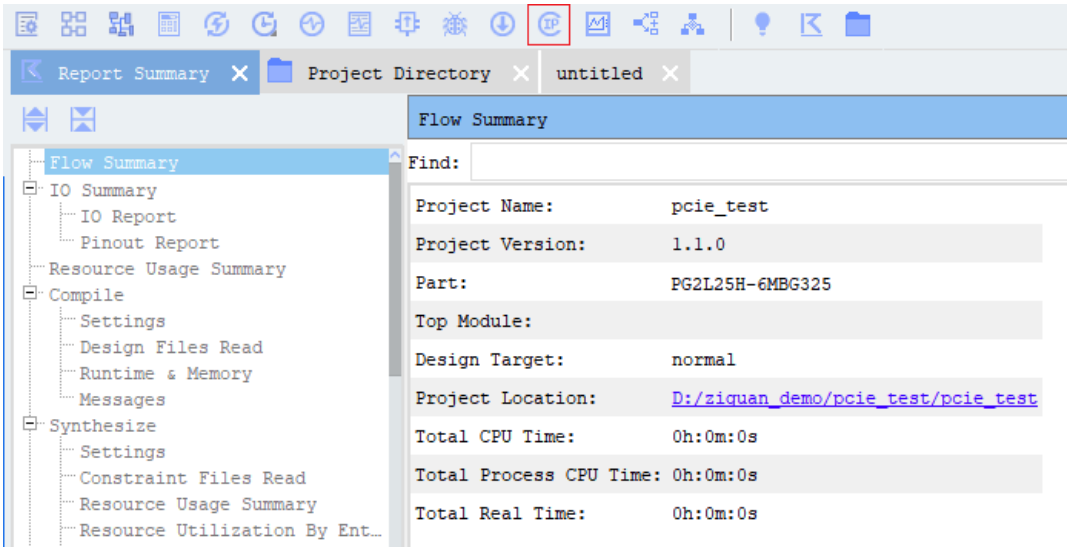


图 14.4-3

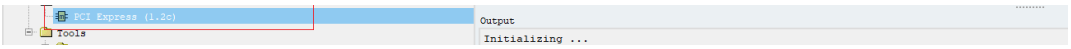


图 14.4-4

选择PCIE IP，取名，然后点击Customize；

在PCIE设置界面中：根据开发板配置lane数，可选择X2，配置参考时钟，可参考下图：

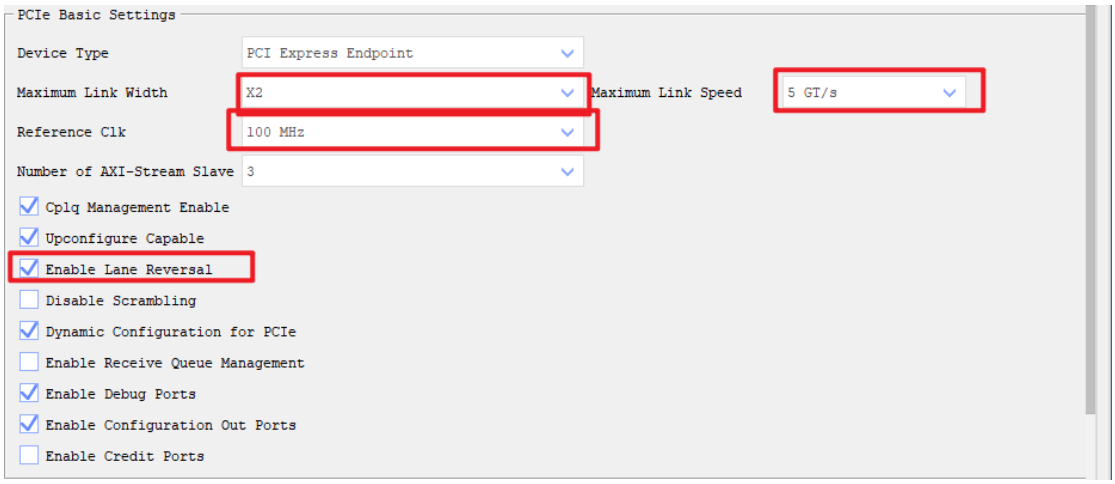


图 14.4-5

需要注意的是，需要勾选上Enable Lane Reversal，否则会导致PCIE实验失败。

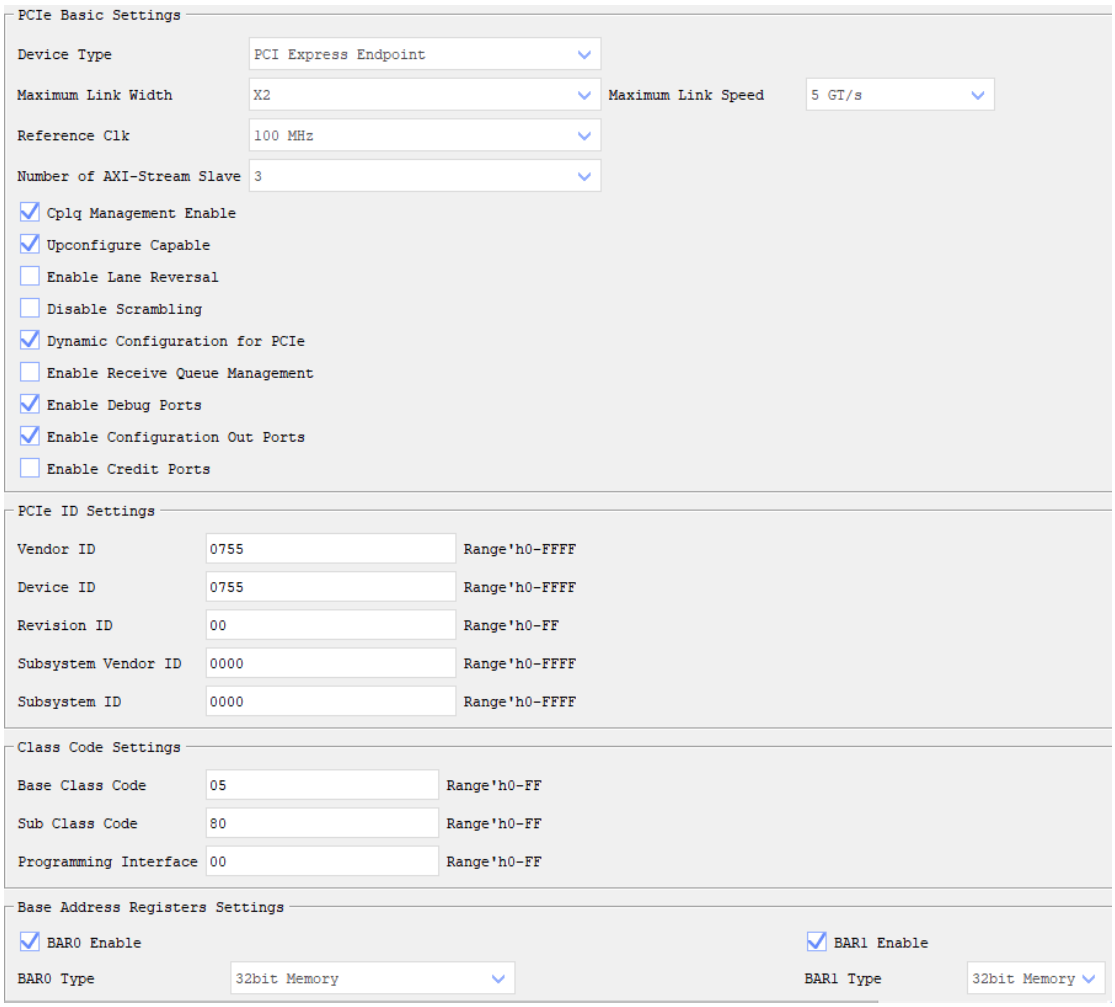


图 14.4-6

其他设置可保持默认，点击Generate生成PCIE IP。

关闭本工程，按此路径打开Example工程：

Xxxxx\pcie_test\ipcore\pcie_test\pnr\example_design

主要:xxxx是自己电脑的路径，后面的pcie_test及其后面的路径是固定的。

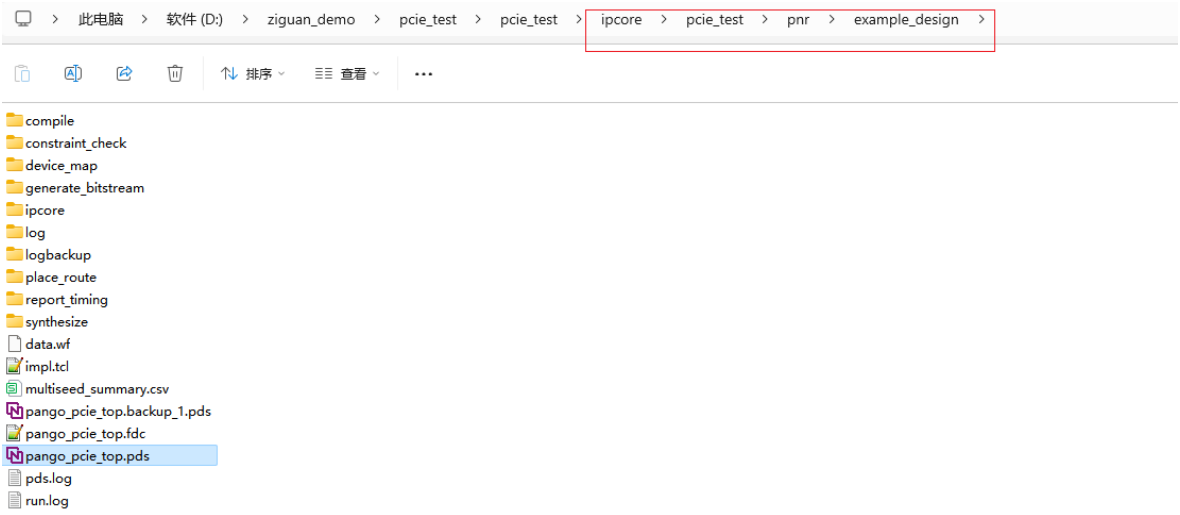


图 14.4-7

按照开发板管脚，修改相关管脚约束：

	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD	DRIVE	BUS_KEEPER	SLEW	OFF_CHIP_TERMINATI
2	txn[0]	OUTPUT								
3	txp[1]	OUTPUT								
4	txp[0]	OUTPUT								
5	rxn[1]	INPUT								
6	rxn[0]	INPUT								
7	rxp[1]	INPUT								
8	rxp[0]	INPUT								
9	pclk_led	OUTPUT	H14	BANKL4	3.3	LVCMOS33	4		FAST	N(default)
10	rdlh_link_up	OUTPUT	E18	BANKL4	3.3	LVCMOS33	4		FAST	N(default)
11	ref_led	OUTPUT	H18	BANKL4	3.3	LVCMOS33	4		FAST	N(default)
12	smlh_link_up	OUTPUT	F17	BANKL4	3.3	LVCMOS33	4		FAST	N(default)
13	txd	OUTPUT	J18	BANKL5	3.3	LVCMOS33	4		FAST	N(default)
14	button_rst_n	INPUT	D10	BANKL4	3.3	LVCMOS33				
15	perst_n	INPUT	L14	BANKL5	3.3	LVCMOS33				
16	ref_clk_n	INPUT	B5	NOBANK						
17	ref_clk_p	INPUT	B6	NOBANK						
18	rxd	INPUT	K18	BANKL5	3.3	LVCMOS33				

图 14.4-8

注意，像txp[0]，txp[1]，rxp[0]，rxp[1]等差分信号都不需要约束。其中txd和rxd是串口。

可按以下方式查看IP核的用户指南，了解Example模块组成；

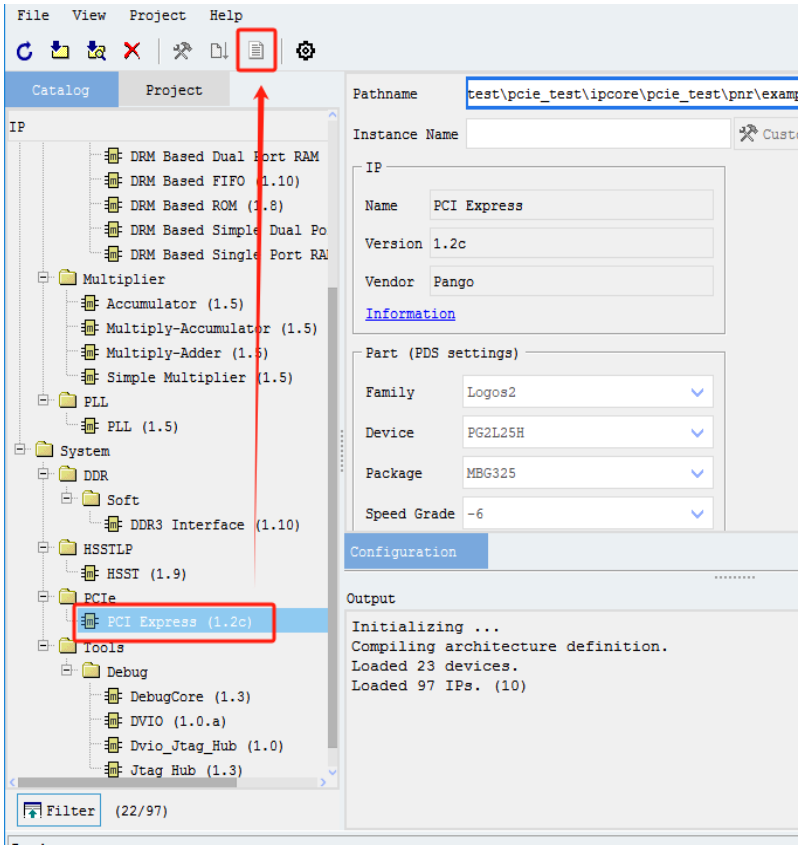


图 14.4-9

14.5. 实验现象

将程序固化到flash内，把开发板插入电脑PCIE卡槽，开机。打开设备管理器，可识别到PCIE设备。

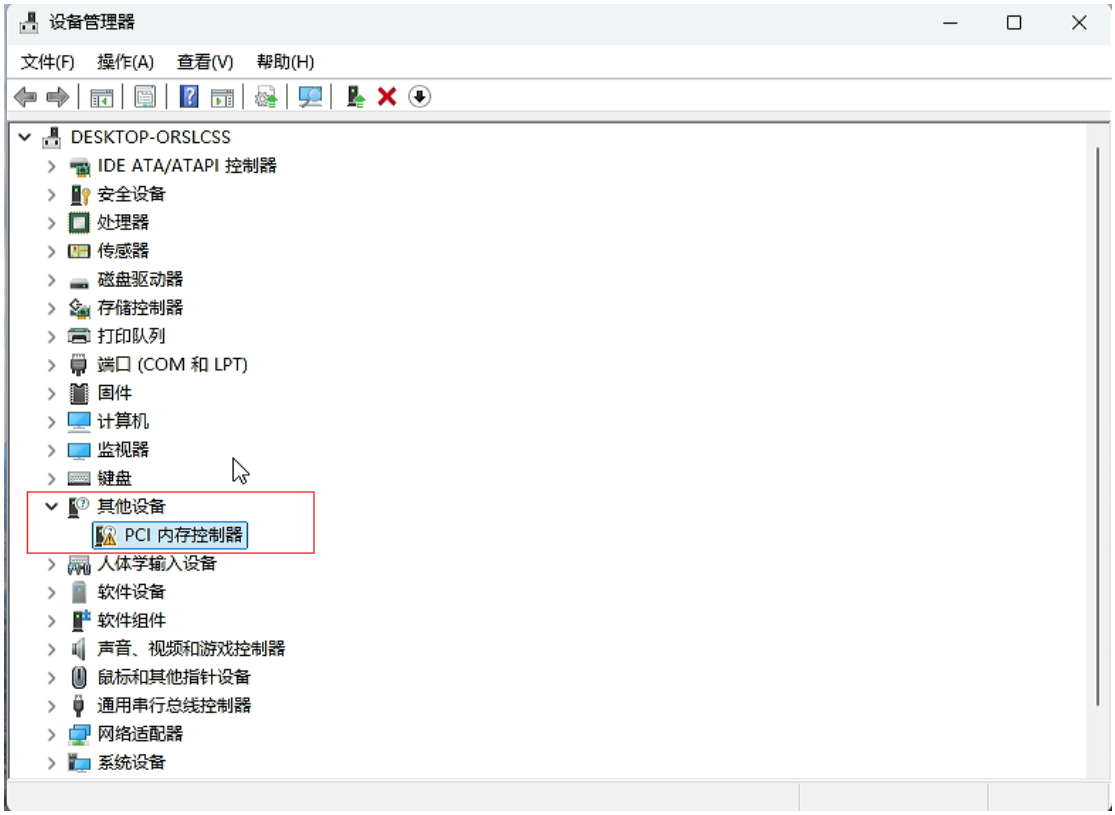


图 14.5-1

Win下能弹出该设备即可。